RL-TR-95-189
Final Technical Report
October 1995

ADVANCED ARTIFICIAL INTELLIGENCE TECHNOLOGY TESTBED

Martin Marietta Corporation

John Zapriala, Thomas Geigel, David Hollingsworth,
Henry Mendenhall, Kenneth Whitebread, Russell Irving,
Michael Wiley, Terry Barnes, Lee Erman, and Dan Kuebler

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-189 has been reviewed and is approved for publication.

APPROVED: Craig S. Anken

CRAIG S. ANKEN
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1995	Final Aug 90 - Feb 95

4. TITLE AND SUBTITLE

ADVANCED ARTIFICIAL INTELLIGENCE TECHNOLOGY TESTBED

5. FUNDING NUMBERS

C - F30602-90-C-0079
PE - 63728F
PR - 2532
TA - 01
WU - 37

6. AUTHOR(S)

John Zapriala, Thomas Geigel, David Hollingsworth, Henry Mendenhall, Kenneth Whitebread, Russell Irving, Michael Wiley, Terry Barnes, Lee Erman, and Dan Kuebler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Martin Marietta Corporation
Advanced Technology Laboratories
Moorestown Corporate Center
Route #38
Moorestown NJ 08057

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory (C3CA)
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-95-189

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Craig S. Anken/C3CA/(315) 330-4833

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes the development and demonstration of the Advanced Artificial Intelligence Technology Testbed (AAITT). The AAITT embodies a structured development paradigm and associated toolkit supporting the design, analysis, integration, evaluation, and execution of large-scale, complex, distributed systems, composed of knowledge-based and conventional components, in the context of various United States Air Force domains, particularly Tactical Commmand, Control, Communcations, and Intelligence. The AAITT's unified modeling, control, and monitoring facilities permit unrelated software components to be integrated without extensive re-engineering by allowing users to easily: (1) configure various application suites; (2) observe and measure the behavior of applications as well as the interactions between their constituent modules; (3) gather and analyze statistics about the occurrence of key events; and (4) flexibly and quickly alter the interaction of modules within the application for further study. Using the AAITT, capabilities neither designed nor originally intended to work together were transformed into integrated problem-solving suites. Four significant demonstrations were successfully conducted and showed that 10:1 integration improvements could be obtained by using the testbed. AAITT development and documentation followed a tailored DoD-STD-2167A process. User and programming manuals, as well as a week-long training course, were developed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Artificial intelligence, Software integration, Testbed, Encapsulation, Distributed system, ABE, Cronus, Code generation	120
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Foreword

The Mission Statement for the Advanced Artificial Intelligence Technology Testbed (AAITT) program called for the AAITT's developers "to specify, design, construct, demonstrate, and document a testbed which will allow its users to (1) easily configure numerous application suites, composed of both conventional and knowledge-based components, by adding, deleting, or intermixing various problem solving modules, (2) observe these modules' actions and interactions, (3) gather and later analyze statistics about the occurrence of key events; and, finally, in response to control strategies, distributions, and allocations in need of improvement, (4) rapidly change the flavor of the interactions among the suite's components for further study."

Where can the motivation for the tool supporting the engineering of software systems described in this mission statement be found? Although examples come to mind in a variety of domains, one need look no further than Command, Control, Communications and Intelligence (C3I) to uncover just such a need to facilitate the development of tomorrow's Command Centers.

Today's Command Centers were planned and built for responding to a threat which no longer exists in its past form. Their design was predicated around the notion of using large, monolithic, single-purpose systems to deal with a similarly large and monolithic enemy. The development of these Centers occurs over an extended period of time. Many of the resulting capabilities are essentially outdated before they are placed in operation, necessitating a constant stream of upgrades. This slow modernization process has not yielded significant improvements because, although more and more electronic systems are being integrated, ongoing advancements in both hardware and software are not being incorporated in a timely fashion. In many cases, new information processing technology which is commonly found within the commercial sector has not found its way into most Command Centers. Unique support infrastructures are subsequently established to overcome each Center's limitations. The consequence is that these Centers are neither adaptable, fully achievable, nor affordable.

Despite their expense, these Centers remain plagued with difficulties. Interoperability is rarely achieved between the multitude of special-purpose systems which have arisen to support today's Command Centers and Commanders. Information and data flow to, and within, these Centers is antiquated, inflexible, and inefficient despite the upgrades being put in place. Resources are being expended to establish custom interchange capabilities which rapidly get out of date. Furthermore, the use of decision aids and tools has not reached the appropriate level, forcing decision makers to continue relying on manual methods.

Tomorrow's requirements only compound the aforementioned problems. Reduced force structures will increase the pressure to "accomplish more with

less." Diminished budgets are forcing duplicate capabilities to be eliminated across the services. Thus, each service will eventually have a mission and assigned responsibilities possessing little overlap with its peers. Regional conflicts, often against new and/or unknown threats, will predominate. These conditions point to the use of Joint Task Forces (JTFs) as the most effective means of response. In addition, given our nation's increasing desire to build coalitions among its many allies, Command Centers will be called upon to go beyond overseeing operations involving multiple services to directing activity combining the forces of various agencies, NATO members, and allied countries. Their respective systems must be accommodated. These coordinated operations will be centered at the location of the JTF CINC (Commander-in-Chief) because the days of "fighting the war out of the Pentagon" are over. Commanders are much better able to assess the situation and select the most appropriate course of action when they have firsthand access to the essential elements of information on which they will base their decisions.

Further complicating these factors is the fact that the occurrence of multiple, simultaneous crises is highly likely. Each crisis may involve a variety of participants, changing levels of hostility, and different locations. We have already witnessed examples where international humanitarian operations have escalated into armed conflicts. Thus, to the US, the relative priorities among these operations will also dynamically change. The fluid nature within and among crises requires the capability to support varying "stand-up levels."

It is foreseen that the military's role in responding to civil disasters at home will also continue to grow. The reasons for being increasingly assigned this type of responsibility are numerous. In addition to a successful record demonstrating their skills in carrying out these emergency relief operations on a world-wide basis, the military is best equipped to meet the challenge on our own soil due to the personnel, command and control structure, and materiel which they already possess. Decreasing budgets will not permit these capabilities to be duplicated between the military and civil response agencies. However, acting in this capacity will require the services to achieve coordination and interoperability with non-military entities, including the aforementioned civil agencies as well as commercial companies willing to lend a hand.

Rapid crisis response, demanding immediate solutions, will be expected, regardless of the situation and any exacerbating circumstances. Systems meeting tomorrow's challenges cannot be pre-determined, pre-assembled, or pre-positioned. Yet the immediacy of each situation challenges us to deftly assemble readily-available and integrated systems without the luxury of time. These systems must utilize and leverage ongoing advances in information technology and provide portable, adaptable, rapidly (re)configurable decision support, integrated to operate in tandem with existing infrastructure capabilities. The level of operational success will be a direct function of the extent to which diverse, necessary, existing computer hardware, communications, data

processing, and decision support capabilities can be identified and rapidly integrated.

The integration of existing hardware and communications systems is certainly an important issue. However, it is an issue already receiving a great deal of attention by commercial vendors implementing the foundations of tomorrow's networks of information systems. Additionally, efforts such as Rome Laboratory's Knowledge-Based Software Assistant program are addressing the need to support and automate the process of developing new software components and systems. However, swiftly establishing, adjusting, and dissolving decision support for the variety of situations which a JTF must face can only be accomplished by providing automated support for flexibly constructing and operating applications consisting of "legacy" (existing) software. There is not enough time to develop solutions from scratch. One must build new solutions by integrating pieces of old solutions. This process requires rapid component incorporation as well as adaptable application assembly. Investing in this type of capability will have a multiplicative effect by fostering the reuse of off-the-shelf components.

Abstracting the process of integrating legacy software into rapid component incorporation and adaptable application assembly is driven by a number of factors.

Legacy software represents both an opportunity as well as an intimidating challenge. A wealth of both conventional and knowledge-based military software and data management capabilities exist. New systems and upgrades are constantly being added to the inventory. On the other hand, some subset of these systems are no longer being maintained. Some of these capabilities already have been used to establish information infrastructures supporting various functional areas, such as Personnel, Intelligence, Operations, Logistics, Plans and Policy, as well as Command, Control, And Communications. Civil crisis response will add the burden of interacting with non-Government organizations and their respective software. Capitalizing on the investment behind these systems is the paramount requirement. Yet, synergistically combining this legacy software and achieving interoperability between them was never considered.

Interoperability can be achieved in many ways and at many levels. However, it is also important to stress that interoperability at the manual process level is unacceptable due to the sheer volume of data which must be captured, processed, and disseminated, as well as the speed at which the overall crisis must be adaptively managed.

Universal data interpreters offer another approach. Here, the goal is to find a way to move data transparently between a variety of data sources and sinks. Success is achieved by identifying a universal representation and constructing interpreters into and out of each supported protocol. Results have proved to be

limited, at best. Common representations are difficult to identify outside of very tightly-bounded domains. The central role which these interpreters play transforms the process into a system bottleneck. Semantic inaccuracies may begin to creep into the translation process and the introduction of even a single new protocol can wreak havoc.

Reimplementation and data migration will, certainly, not provide the solution for these requirements. This process is, at best, prohibitively expensive, and, at worst, impossible to achieve. The capabilities offered by legacy software must be transparently available within their native user environments. Otherwise, decision makers will be faced with a learning curve offering lessened productivity and possibly counterproductive errors.

The aforementioned approaches strive for precise solutions. Thus, they frequently cannot be achieved within the crucial deadlines which must be met to successfully provide dynamic crisis response. Furthermore, the best or most elegant answer may be neither possible nor desired. A sufficient solution may be the most worthwhile due to the brief period of time during which any given decision support configuration remains valid.

The solution embodied within the notion of Rapid Component Incorporation calls for the use of standard, customizable, automatically-generated, reusable, control and information-exchange "adapters" for both new as well as existing components and systems. These adapters facilitate interaction with other similarly-equipped modules.

The resulting modules, each of which provides an answer for a discrete portion of the entire solution, must be subsequently transformed into a cohesively operating package through the second step of the process, Adaptable Application Assembly. This notion implies a number of capabilities because one is faced with the challenge of taking a set of building blocks and turning them into a properly interacting whole. Thus, users must be able to specify how each piece will act; identify how the pieces will cooperatively interact; and, most importantly, gain insight into and understand why the resulting system does not operate exactly as required, desired, or envisioned so that corrective action can be taken.

Successfully implementing Rapid Component Incorporation and Adaptable Application Assembly permits a "divide and conquer" strategy to be employed in realizing tomorrow's distributed decision support systems. This strategy is congruent with the fact that large, ready-made, one-of-a-kind systems do not offer the required level of agility. Dividing and conquering does require candidate components, capable of satisfying the specified requirements, to be initially identified and subsequently integrated. However, these disadvantages are far outweighed by the advantages of this approach.

Dividing and conquering allows distributed application developers to only construct what is needed. The resulting system is not encumbered with extraneous, complicating capabilities. Each subproblem is addressed using the most appropriate software and hardware paradigms. Solutions are not force-fitted into a constrictive, homogeneous approach. Furthermore, technology upgrades are incorporated into the solution as they become available with minimal disruption. Alternate approaches are swapped in or out to facilitate component evaluation and solution improvement. A total reconstruction is not required each time the situation requires that functionality be added or subtracted. More importantly, component reuse is greatly facilitated. Finally, this philosophy promotes investigations into competing interaction and control approaches allowing continuous solution refinement to occur.

The potentially important role of the AAITT in facilitating tomorrow's distributed, decision support systems, such as those found within JTF Command Centers, and the power wielded by a user able to effect these systems by dividing and conquering served to focus the development efforts throughout this program.

The AAITT takes a just place among the limited ranks of tools which have truly advanced distributed systems technology. By building on the paradigm of "Programming in the Large" established by Cimflex Teknowledge's ABE™ product, the AAITT embodies a graphical Modeling, Control, and Monitoring methodology and associated toolkit to facilitate heterogeneous component integration. With the testbed, users have realized large productivity gains (> 10x) in the tasks of Rapid Component Incorporation and Adaptive Application Assembly.

By exploiting research in distributed computing, module-oriented programming, and object-oriented simulation, a disciplined technique for integrating both knowledge-based and conventional software components is now available. The testbed allows the performance of these integrated applications to be measured at various levels of granularity using substantial instrumentation. The result? Distributed application developers arrive at high-quality solutions through low-cost experimentation.

The AAITT has been used to successfully construct applications within several domains by its developers. It is time for it to be examined by more distributed system builders as a possible paradigm shift in their work. The AAITT needs this trial to further gauge the effectiveness of its capabilities, the power behind some of which has yet to be discovered.

Russell E. Frew, LTC., USA (Ret.)
Director, Artificial Intelligence Laboratory
Lockheed Martin Advanced Technology Laboratories

Contents

Figures

Tables

1 Summary

Over the past 4.5 years, a team consisting of Lockheed Martin Advanced Technology Laboratories, Teknowledge Federal Systems, GE Corporate Research and Development, and dek M&TS have developed and demonstrated the Advanced Artificial Intelligence Technology Testbed (AAITT). This laboratory testbed embodies a structured development paradigm and associated toolkit to support the design, analysis, integration, evaluation, and execution of large-scale, complex, distributed software systems, composed of knowledge-based and conventional components, in the context of various USAF (United States Air Force) domains, particularly Tactical C3I (Command, Control, Communications, and Intelligence).

The AAITT's unified modeling, control, and monitoring facilities permit unrelated software components to be integrated without extensive re-engineering by allowing users to easily (1) configure various application suites; (2) observe and measure the behavior of applications as well as the interactions between their constituent modules; (3) gather and analyze statistics about the occurrence of key events; and (4) flexibly and quickly alter the interaction of modules within the application for further study. Thus, capabilities neither designed nor originally intended to work together can be transformed into integrated problem-solving suites composed of intelligent information technologies, diverse functional specialties, "best-of-breed" applications, off-the-shelf software, and, significantly, legacy systems.

The underlying need for such a distributed system development, test, and evaluation environment is both powerful and ever increasing. The role and complexity of the various decision aids that will be prevalent throughout every aspect of both military and commercial operations will continue to grow in the future. These decision aids will be solving large, decomposable problems within intricate and data-rich domains. They will be composed of both knowledge-based and conventional software modules interacting as part of some predefined problem-solving strategy. The ability to iteratively build, cost-effectively integrate, and most importantly, deploy these multi-agent applications suites will depend on the existence of a facility in which these suites can be studied under an "electronic microscope," such as the AAITT, so that they can be better understood.

The design of the AAITT was driven by the desire to provide comprehensive support for Component Embedders, individuals tasked with transforming stand-alone components into testbed-compliant modules; Application Architects, users responsible for taking multiple modules and assembling them into distributed applications; and Component Evaluators, who are interested in analyzing the behavior of components along multiple dimensions. A key concept within the testbed was that "wrappers" would be used to envelop components and effect communications as well as control.

To realize the testbed, core Simulation and Database capabilities were teamed with a Modeling, Control, and Monitoring workstation (MCM). These items communicate and interact via a Distributed Processing Substrate (DPS). Together, they constitute the AAITT.

The MCM supports modeling, offering users the ability to construct a graphical representation of an application's solution strategy. The workstation also acts as the testbed's control panel, permitting users to start, stop and suspend application or module execution. Finally, the MCM possesses measurement, instrumentation, and monitoring capabilities to support the synthesis, analysis, and evaluation of AAITT-resident applications.

The DPS provides distributed processing and communication capabilities that support the integration and concurrent execution of multiple, independent, knowledge-based and conventional software components across heterogeneous computing systems.

Testbed development and documentation followed a tailored DoD-STD-2167A process. The AAITT was validated through three separate Formal Qualification Tests, conducted to assess the software's compliance with identified qualification requirements. In addition, user and programming manuals, as well as a week-long training course were developed.

The completed AAITT offers a wide range of tools and capabilities, including a graphical approach to configuring, encapsulating, and integrating components; customizable, automatically-generated component wrappers providing immediate productivity gains; application monitoring at several levels, from resource usage to solution quality; and a user-customizable graphic interface for control and instrumentation. These features permit unparalleled advances in distributed system construction, such as the accelerated establishment of interoperable suites of scalable software tools; rapid system prototyping centered around adaptive component composition and reuse; as well as metric-based architecture and component assessments.

Three significant demonstrations were successfully conducted during the course of the base program. These events presented the testbed's ability to, respectively, integrate and execute a basic, distributed suite of knowledge-based and conventional components; expand the basic suite into a large-scale application; and support domains other than Tactical C3I. A fourth demonstration, completed under the auspices of an engineering change, showed the AAITT providing effective support for performing "Instrumented Domain Experiments."

The program's demonstrations repeatedly underscored the testbed's role in decreasing the software integration costs associated with distributed, heterogeneous applications. The second, Large-Scale Demonstration, offered evidence worthy of emphasis. A distributed system composed of nine

independent, contractor- and Government-developed components was integrated, debugged , and executed in only 25 days using the AAITT. In the absence of the testbed, a similar development effort on the ARPA/AFWL (Advanced Research Projects Agency/Air Force Wright Laboratory) Pilot's Associate program required approximately 250 days — use of the AAITT yielded a 10:1 integration improvement.

2 Introduction

This section presents the impetus for the Advanced Artificial Intelligence Technology Testbed program, related work existing at the time the effort was initiated, the team which was assembled, and the organization of this document.

2.1 Motivation

On 9 March 1990, the Rome Air Development Center (now called Rome Laboratory) issued the AAITT Request for Proposal (RFP). The Statement of Work contained in the RFP described the state-of-the-art in this area as follows:

> *"Over the past several years RADC has been conducting research and development (R&D) in various subdisciplines of Artificial Intelligence (AI) technology and in the development of applied systems that embody AI technology. Typically, the research projects have been focused in specific areas (reasoning with uncertainty, planning) and have used diverse problem domains as a context for the research. Similarly, application projects have typically addressed a sub-function of some larger problem domain (C3 Counter Measures Battle Management Decision Aid as a sub-function in a Tactical Air Control System). Fielding robust systems that embody AI technology requires integration of several technical approaches, integration of subsystems, and scaling up of the technology in dynamic, complex, and time-constrained military environments. Although there are several research projects involving integration and scaling of AI technology, in general there is a lack of fundamental understanding in this area. Several approaches and solutions have been attempted, but an optimal or even sufficient solution is usually not known a priori and often not found until the second or third implementation. We believe an experimental approach is dictated. The testbed developed under this effort will serve as a vehicle for the integration, analysis, design and evaluation of large (primarily multi-agent) knowledge-based systems in complex military problem domains."*

This description was referring to the need for an environment that would allow both related and unrelated software components to be assembled into more complex systems than could currently be easily constructed. This environment would be called a testbed because it would allow for the rapid (re)configuration of these systems and would allow them to be instrumented so that their architectures and behaviors could be better understood.

2.2 Relevant Work

Several efforts were cited for their relevance within the RFP. The first was COPES (Cooperating Expert Systems), *"... an analysis and design project to*

develop an architecture to support [the] integration of three specific decision aids to form a system of cooperative agents." This work emphasized the use of "wrappers" to act as intermediaries for component communication. The second project noted was ABE™ (A Better Environment), a software system to support the design and development of intelligent systems using module-oriented programming. ABE™ provided an excellent foundation for developing a multi-module application development environment. The third program mentioned was TAC-2, a Testbed For Integrating Cooperating Knowledge-Based Air Force Decision Aids. Under this effort, an architecture to support the integration of loosely coupled decision aids, based on the notion of a centralized router, was implemented.

2.3 Team Members

Given this background, a combined team from the Lockheed Martin Advanced Technology Laboratories (ATL, formerly GE Aerospace Advanced Technology Laboratories), Teknowledge Federal Systems, and dek Marketing & Technical Services, proposed to build an environment to advance the state-of-the-art in distributed application development. Among their strengths, the team cited extensive experience in the development of distributed multi-agent systems. In particular, members of the team had collective responsibility for both the System Status module during Phase One of the Air Force Wright Laboratory Pilot's Associate (PA) program as well as the development of realtime system transition tools for that effort. In addition, ATL was also the prime contractor for ARPA's Submarine Operational Automation System (SOAS). Teknowledge Federal Systems possesses some of industry's leading authorities on distributed AI control strategies. USAF-related domain expertise would be provided by dek Marketing & Technical Services. Due to an internal transfer of personnel following award, GE Corporate Research and Development subsequently joined the original team.

The proposed testbed would facilitate the recurring process of configuring application suites by permitting users to (1) add, delete, or intermix various problem-solving modules; (2) observe the nature of the modules' actions and interactions; (3) gather and, later, analyze statistics about the application, its constituent modules, and the occurrence of key events, to pinpoint control strategies and module-to-host assignments in need of improvement; and (4) rapidly change the flavor of the interactions among the suite's components for further study.

Under the base program, three applications were developed and delivered to demonstrate the feasibility, scalability, and reusability of the resulting testbed, which is built atop ABE™ and greatly expands upon the "wrapper" concept described in the COPES report. Initially, the TAC-2 application was re-implemented within the AAITT to show feasibility. Next, the size of this application was tripled via the inclusion of additional, independently-developed

components to demonstrate scalability. Finally, an application composed of planning- and scheduling-related components was developed to substantiate reusability.

2.4 Report Organization

This Final Report is intended to provide a complete picture of the Advanced Artificial Intelligence Technology Testbed program. The need for and potential operational use of such a testbed was discussed earlier within the Foreword. An executive-level description of the entire effort can be found in Section 1, Summary. The current, introductory section has set the stage by focusing on various pre-award topics.

The organization of the remainder of the Final Report is described below.

Section 3, AAITT Definitions, Roles, and Requirements, presents the team's view of both the issues to be addressed and the problems to be solved by this effort as well as its initial approach to defining the AAITT's functionality. Section 4, The AAITT Architecture, provides an overview of the solution which was realized by the team. Section 5, AAITT Features, describes the testbed's major attributes. Section 6, AAITT Applications, discusses the scenarios and underlying architectures of the three major applications constructed to demonstrate the AAITT's capabilities under the base program. Section 7, Where to Find More Information, offers readers a road map to related documents providing amplifying data about the effort. Section 8, Results and Discussion, enumerates the contract's broad range of results for both the technical community as well as operational users. Section 9, Conclusions, presents the team's perspective on what was accomplished. Section 10, Recommendations, suggests a number of judicious extensions to the testbed which would further increase its value to the USAF technical and operational communities.

Appendix A, Instrumented Domain Experiments (IDEs), documents the team's efforts to investigate and demonstrate the AAITT's ability to support IDEs, conducted under the auspices of an engineering change. Finally, the Glossary as well as List of Acronyms and Abbreviations will aid readers in understanding this Final Report.

3 AAITT Definitions, Roles, and Objectives

It was important to standardize terminology from the project's outset. Thus, the following definitions were adopted. Remembering these terms will greatly aid in understanding much of this Report.

Component	A stand-alone conventional or knowledge-based program.
Component Interface Manager	An encapsulating software element, or wrapper, responsible for managing the interface between a component and the remainder of the testbed.
Module	A (possibly modified) component and associated component interface manager.
Application / Subapplication	An assemblage of subapplications and/or modules.

Building on these definitions, the testbed team then considered, "Who are the testbed's users and what will they want to accomplish?" This led to the term 'user role.' The inclusion of the word 'role' is an important distinction because a single individual may, at various times, operate within one or more of these roles. The following user roles were identified:

Testbed Developers	The builders of the testbed software, to include the original program team as well as the eventual inheritors of the AAITT, responsible for maintaining and extending it following delivery to Rome Laboratory.
Component Developers	The builders of the knowledge-based software or conventional components which can be used as the constituents of a testbed application. These individuals may require some level of support from the testbed in cases where these components will be completely, or largely, developed within the testbed. On the other hand, in the case of legacy or "off-the-shelf" components, these developers will not be AAITT users.
Component Embedders	These individuals create AAITT-compliant modules from stand-alone components using their detailed understanding of the

testbed's control and communication protocols as well as varying levels of knowledge about the components which they are embedding. Possession of an in-depth awareness of the target application architecture is not always necessary within this role.

Application Architects..... These users define the multi-module application architecture by identifying the application's constituent modules and subsequently specifying inter-module connectivity and interaction. This role does not require a detailed understanding of the AAITT's underlying communication layer. Instead, the focus is on application-level modeling, control, and monitoring. Application Architects are the most likely to be found conducting demonstrations.

Component Evaluators..... This user role determines how well one or more of an application's modules are operating along a variety of dimensions. The scope of these evaluations can range from simple issues such as resource utilization to complex domain-specific questions such as solution quality.

Application Fielders..... These individuals take a completed, testbed-resident application and generate a strategy for porting it to a target computer configuration which may or may not include the AAITT. Accomplishing this task would require modeling the processing and communications resources available within the target configuration before conducting trade-off analyses to compare application architecture alternatives.

Application Users..... These customers for fielded applications are AAITT users only in cases where the Application Fielder made the decision to include the testbed, or some portion of it, as an element of his/her fielding strategy.

After identifying these user roles and understanding their respective needs, the decision was made to place the most emphasis on supporting Component Embedders, Application Architects, and Component Evaluators. This decision was predicated on the fact that, at least initially, these three roles would be the predominant users of the testbed. It was envisioned that the needs of the other roles could be even better understood and more effectively addressed after several applications were successfully constructed within the confines of the testbed. This is not to say that the needs of other roles would be forgotten. In fact, the AAITT's initial, core set of capabilities would provide a substantial level of support for these other roles as well. However, in many cases, the needs of these roles would be met implicitly. For example, the Application User would still have the powerful ability to execute and control often unwieldy, distributed applications using the testbed.

Figure 1 summarizes the relationships between the aforementioned user roles, the AAITT, components, modules, as well as the resulting applications and solutions produced using the testbed.

Next, the types of questions which these user roles might pose were considered. These envisioned user questions included:

- *"How do I load the application?"*
- *"What is consuming most of my available resources?"*
- *"Why aren't these modules talking to each other?"*
- *"How is the application performing?"*

A large matrix was subsequently constructed. The rows consisted of the aforementioned user roles. Their questions became the columns. Whenever a question pertained to a particular user role, the intersecting matrix position was checked off. In many cases, a single question was relevant to several users. The completed matrix became the starting point for defining required AAITT functionality beyond the initial requirements stated earlier. The satisfaction of these requirements became the organizing principle for the Formal Qualification Tests used to validate the testbed software.

Seven top-level requirements established the project's overall direction:

1. The testbed will provide run-time communications facilities for multi-component applications.
2. The testbed will support components running in heterogeneous hardware and software environments.

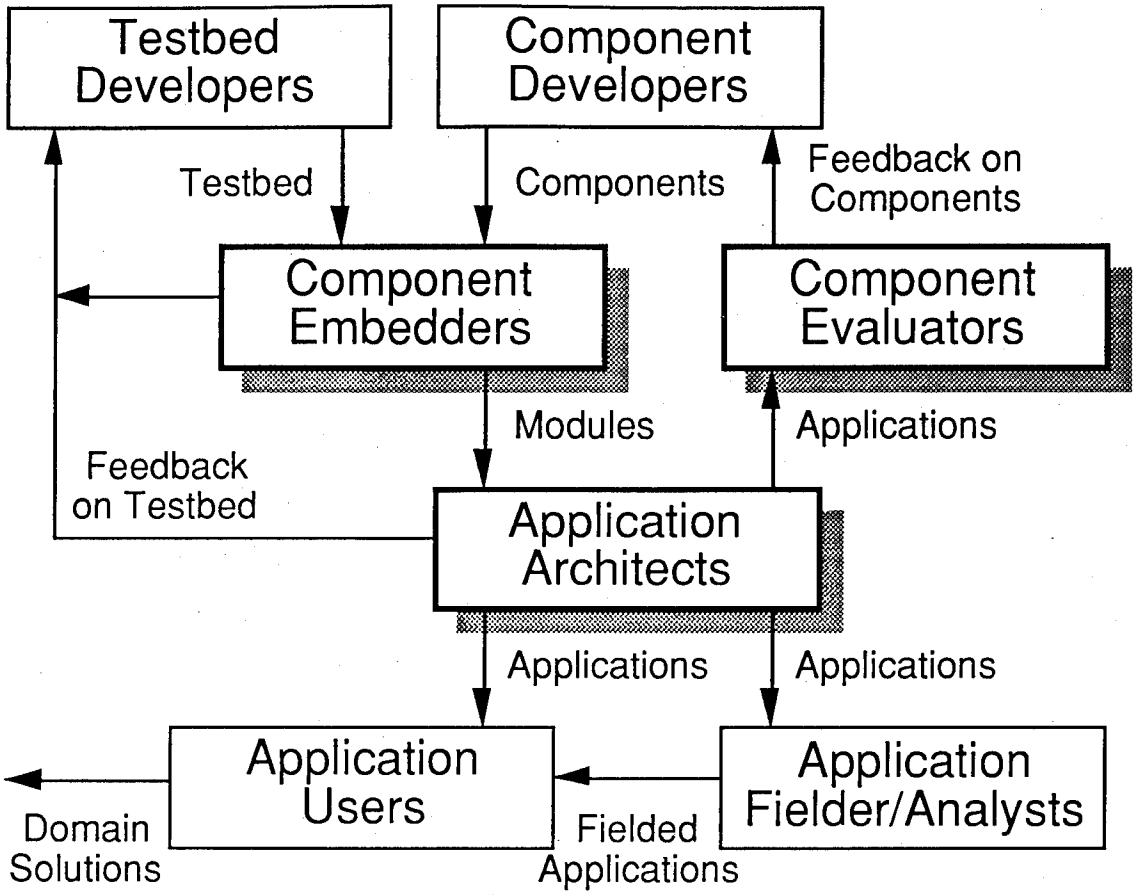


Figure 1. Testbed User Roles and Relationships

3. The testbed will support the concurrent execution of the multiple components.

4. The AAITT testbed will support and facilitate the process of assembling and integrating multi-component applications.

5. The testbed will provide a run-time control mechanism for multi-component applications.

6. The testbed will provide run-time monitoring, as well as measurement and data logging to support post-processing performance analysis.

7. The testbed will contain core Simulation and Database modules.

The full set of AAITT requirements are recorded in the AAITT Software Requirements Specification. Information on obtaining this document can be found in Section 7, Where To Find More Information.

4 The AAITT Architecture

The AAITT program's design effort commenced after the completion of the testbed's requirements specification phase. This section presents an overview of the phase's results and begins with a discussion of the AAITT's three major elements. Two additional architectural elements normally required within a testbed-resident application are subsequently covered. Finally, this architectural overview is concluded with a brief narrative of the manner in which distributed applications reside atop the testbed. Amplifying information can be found within the AAITT Software Design Document.

4.1 Major Architectural Elements

The top-level AAITT architecture is shown in Figure 2. Although a typical AAITT Application is included as part of the diagram, the intent here is to focus on the testbed's three major elements:

1. Distributed Processing Substrate (DPS),
2. Modeling, Control, and Monitoring Workstation (MCM), and
3. Core Simulation (LACE/ERIC) and Database (TAC-DB) modules.

Each of these major elements is described below.

4.1.1 Distributed Processing Substrate

The DPS offers distributed processing and communications capabilities that support the integration of multiple, independent, conventional and knowledge-based software components executing concurrently across heterogeneous computing systems. It also provides reliable, transparently-routed inter-module communication and supports the MCM's control and monitoring requirements.

The Distributed Processing Substrate is composed of an underlying Distributed Processing and Communication System residing between the AAITT Protocol layer and TCP/IP. This intermediate element, a non-developmental item, was realized using BBN's Cronus distributed computing environment, following an analysis which compared six candidates using a set of pre-defined criteria as well as additional qualitative measures. In general, the AAITT Protocol layer defines both module-module and module-MCM communication. It also defines, among other things, how modules are to log data, suspend execution to effect breakpointing, and respond to control directives from the testbed. The protocol consists of 25 separate application, state transition, logging, breakpoint and CIM (Component Interface Manager)-query operations.

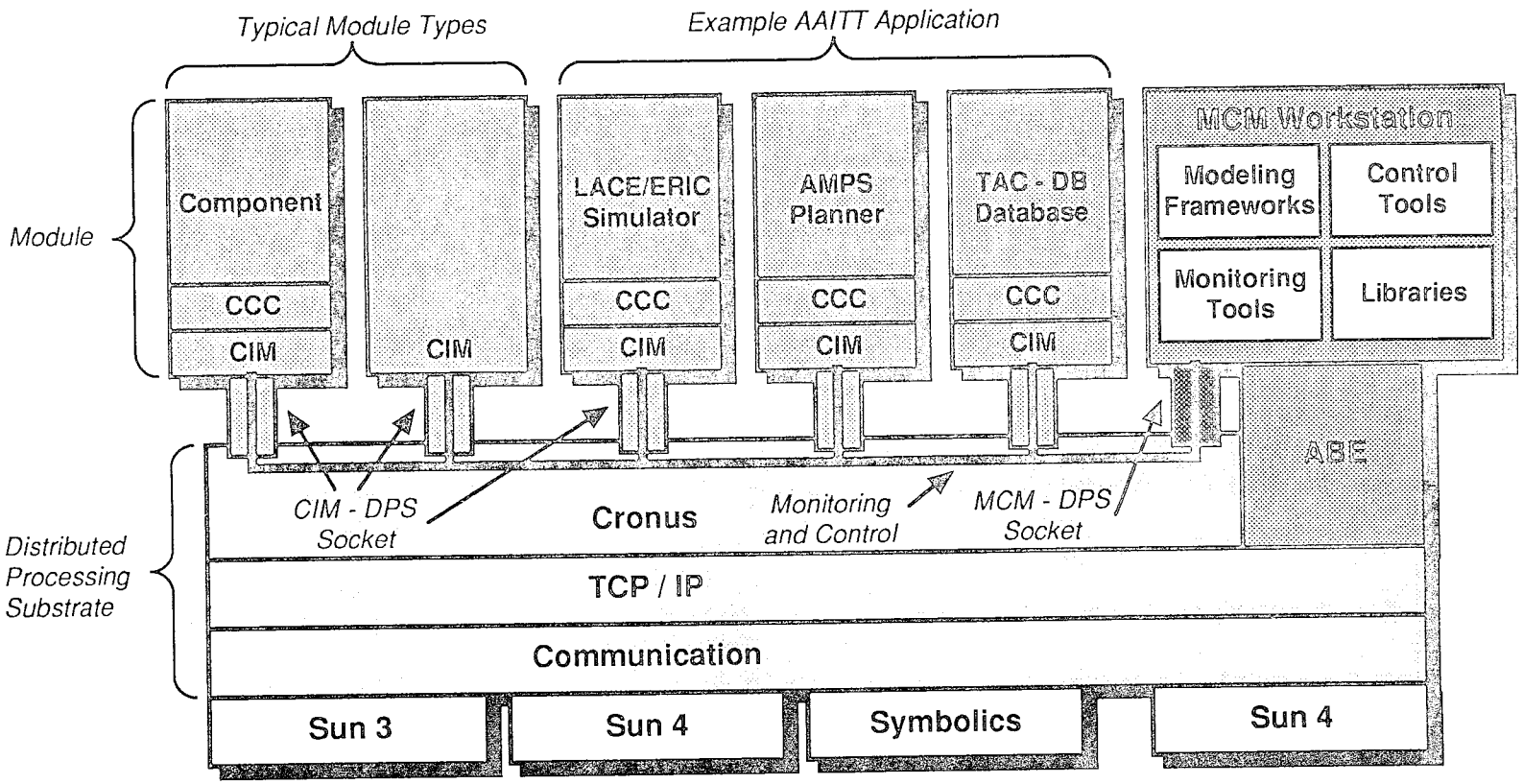


Figure 2. Top-Level AAITT Architecture

4.1.2 Modeling, Control and Monitoring Workstation

As its name implies, the MCM Workstation is an integral testbed-resident facility that provides the means to model, control and monitor applications. Thus, the workstation supports the construction of a graphical model that represents the application's solution strategy; acts as the testbed's control panel, permitting users to start, stop, as well as suspend and resume execution of the application via the same graphical model; and possesses measurement, instrumentation and monitoring capabilities to facilitate the synthesis, analysis, and evaluation of large-scale, multi-agent, distributed applications executing within the testbed. In this context, a system is monitored to collect measurements, which are analyzed by the application developer via instrumentation.

Both modeling and the display of measurement information are accomplished at the MCM Workstation. However, as shown in the diagram, all communication between the MCM and an application's constituent modules for control, status, as well as logging purposes conforms to the AAITT Protocol and utilizes the testbed's Distributed Processing Substrate as a conduit. The MCM was built by adding Datatype, Module and Application "Frameworks" atop Cimflex Teknowledge's ABE™ system to support the MCM's modeling needs. Extensive use of ABE's existing monitoring and metrics analysis capabilities was also made.

The MCM Workstation was originally hosted on a Symbolics™ special-purpose workstation. As shown in Figure 2, the MCM was later ported to a Sun™ general-purpose computer under the auspices of an engineering change.

4.1.3 Core Simulation and Database Modules

The testbed's core simulation and relational database capabilities are implemented as integral, testbed-resident modules, that can be, but do not necessarily have to be, constituents of any application constructed within the testbed (either individually or together).

Both modules were Government-furnished software packages. The first provides a generic capability for simulating objects and events within USAF C3I problem domains using the Rome Laboratory (RL)-developed ERIC/LACE simulator. The second offers an integrated and shared repository for the representation and maintenance of domain-specific data using the RL-sponsored TAC-DB database, acting as an information source for an application's other modules. This Oracle-based database supports SQL (Structured Query Language) and assists in the realization of C3I applications by managing unclassified intelligence data about various friendly and enemy units, equipment, and installations.

Table 1, below, provides a summary mapping between the AAITT's top-level requirements and its Distributed Processing Substrate (DPS); Modeling, Control, and Monitoring Workstation (MCM); and Core Simulation and Database (Sim/DB) elements.

Requirement	DPS	MCM	Sim/DB
1. The testbed will provide run-time communications facilities for multi-component applications.	√		
2. The testbed will support components running in heterogeneous hardware and software environments.	√	√	
3. The testbed will support the concurrent execution of the multiple components.	√	√	
4. The AAITT testbed will support and facilitate the process of assembling and integrating multi-component applications.		√	
5. The testbed will provide a run-time control mechanism for multi-component applications.		√	
6. The testbed will provide run-time monitoring, as well as measurement and data logging to support post-processing performance analysis.		√	
7. The testbed will contain core Simulation and Database modules.			√

Table 1. Mapping between Top-Level Requirements and Major Elements

4.2 Additional Architectural Elements

Fully realizing an application normally requires the inclusion of two additional architectural elements as part of each module — a Component Interface Manager (CIM) and associated CIM-to-Component-Communication (CCC) strategy. Figure 3 presents an exploded view of an example Module; its

constituent CIM, CCC, and Component; as well as the various Operations which form the AAITT protocol. CIM and CCC overviews are presented below.

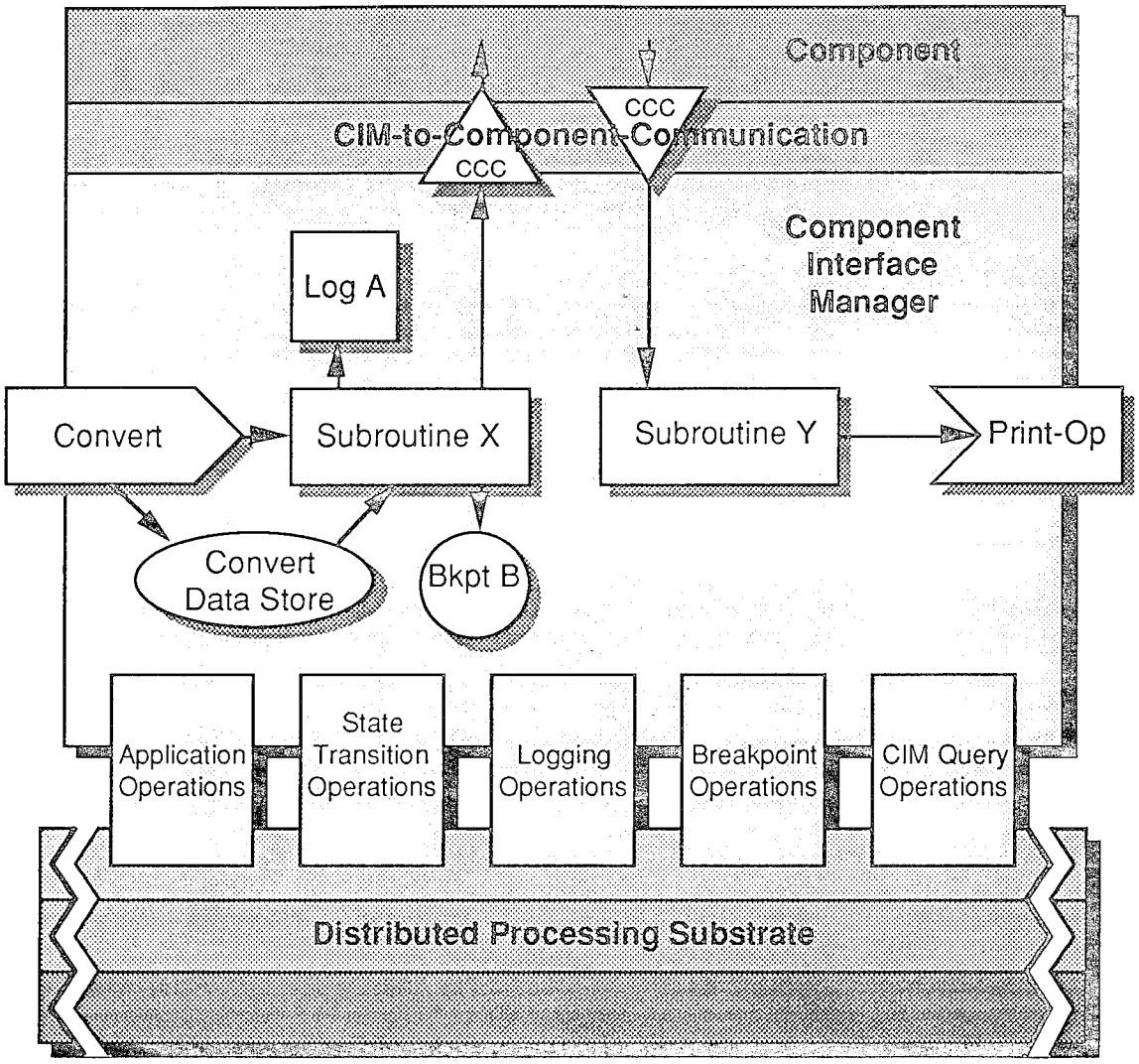


Figure 3. Exploded View of Module Constituents

4.2.1 Component Interface Managers

In order for a software component to be embedded into the testbed, a model of how the component will interact with other modules must be constructed. This model of interaction will subsequently be embodied within an AAITT

Component Interface Manager for that component. The CIM is analogous to the more commonly used term "wrapper."

Each component's attendant CIM is specified graphically via the MCM's Module Framework and includes all of the module's inputs and outputs as well as any additional preprocessing and postprocessing steps which must be completed before data can be presented to or received from the component, respectively. The interconnected icons shown in Figure 3 are representative of a graphical model generated by a Component Embedder. The testbed's code generation capabilities transform the graphical model into executable code.

Within the AAITT, the module's inputs and outputs are referred to as "ports." The CIM also contains code or "port bodies" for each of the ports that have been defined for the module. In addition to ports, the module's graphical model supports the definition of data structures ("data stores" in AAITT terminology), subroutines, CIM-to-Component-Communication (CCC) interfaces, breakpoints, and logging taps. The inclusion of each of these objects is subsequently reflected as additional functionality within the automatically-generated CIM.

4.2.2 CIM-to-Component-Communication

A critical aspect of every AAITT application is the communication mechanism employed between a component and its associated CIM. A number of CCC strategies are presently supported within the AAITT, such as:

- UNIX Sockets
- Files (Pipes)
- Cronus
- Direct Subroutine Call (LISP-language only)
- Queue Variables (LISP-language only)
- Other Inter-Process Communication, including shared memory or message passing.

The MCM's modeling mechanism provides the ability to isolate the CCC portion of a CIM to aid in its development and analysis. In addition, a generic, socket-based CCC library for UNIX-hosted components is available within the AAITT.

4.3 AAITT Applications

AAITT applications execute atop the testbed. Applications are configured as one or more modules. Modules are shown as the upright elements within Figure 2. For each module, its CIM acts as the interface between the module and the remainder of the testbed, including peer modules and the MCM. Within the CIM, the CCC acts as the bridge between the CIM and the component.

Generally, applications can be composed of two module types. Core modules offer capabilities frequently required within many C3I-centered problem-solving

suites. Optional modules can be embedded into the testbed at the discretion of the Application Architect to provide additional functionality. Once a baseline application is created, supplementary architectures can be easily constructed, tested, and catalogued with minimal modification to support experimentation. As detailed in Appendix A, this process was used to facilitate the conduct of an example Instrumented Domain Experiment using the AAITT.

5 AAITT Features

The AAITT's features are best used at the MCM Workstation. Users access it to model applications, compile them, specify host assignments for their constituent modules, as well as execute, monitor, and analyze their performance. The MCM Workstation acts as a control panel, permitting users to start, stop, as well as suspend and resume application execution via its graphical model. The Workstation also provides measurement, instrumentation and monitoring capabilities to facilitate the synthesis, analysis, and evaluation of large-scale, multi-agent, distributed applications executing within the testbed.

As its name implies, the MCM Workstation provides Modeling Tools, Control Tools, and Monitoring Tools. This section is organized in the same manner.

5.1 Modeling Tools

The MCM Workstation's Modeling Tools facilitate the graphical model construction process via the Application, Module, and Datatype Frameworks. An integral Catalog System is used to store and reuse these models.

The Application Architect uses the Application Framework to retrieve cataloged module models for ensuing inclusion within an application. The Framework is also used to specify the links between these modules; make default module-to-host assignments; and save the completed application in the catalog.

The development of graphical models for each of an application's modules is accomplished by the Component Embedder using the Module Framework's capabilities. These models can be added to the catalog and subsequently made available to the Application Architect.

The Datatype Framework is used by the Component Embedder to define the set of basic datatypes, called "cantypes" (a contraction of "canonical types"), used within an application. Named sequences of required and optional cantypes are then assembled into "signatures" using the framework. Signatures define the data communicated in to, within, and out of a module.

The Catalog System serves as a library manager for completed applications, modules, and datatypes. Previously constructed applications, modules, and datatypes can be adapted or reused to create new, or variations of existing, applications, modules, or datatypes.

5.1.1 Application Framework

The Application Framework (AF) is a tool used by the Application Architect to construct new AAITT applications. Figure 4 shows a nine-module application, graphically modeled in an Application Framework window. Also shown is the Application Operations menu.

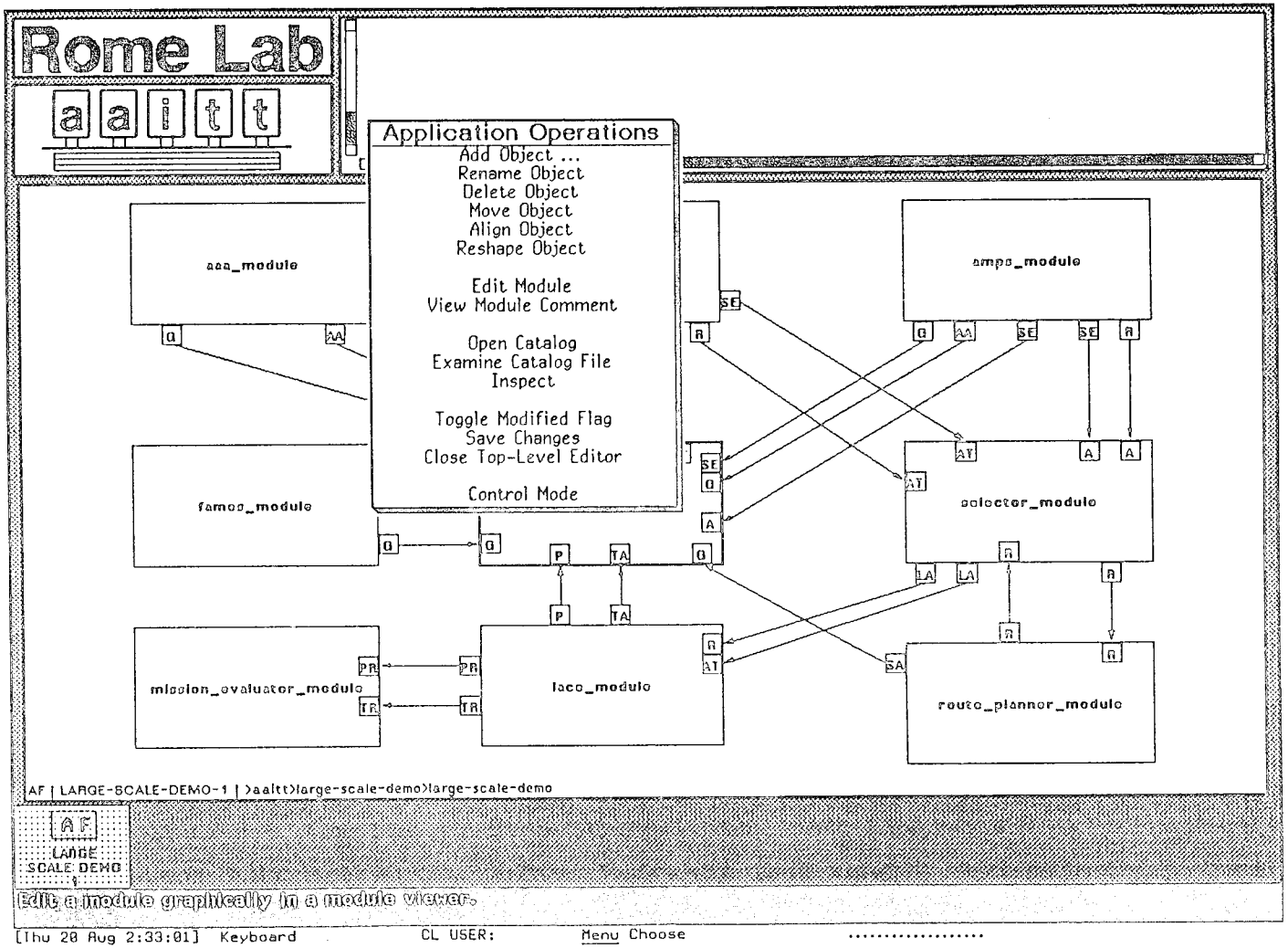


Figure 4. The Application Framework

The Application Architect starts with a concept of operations for the application and a catalog of modules, i.e., embedded components, and graphically produces a module configuration which describes the desired application. The result is not the application itself, but a model of the application from which the actual application can be constructed via compilation. It is important to note that the MCM Workstation uses this same graphical application diagram as the interface for the Application Architect or Application User to execute and control the application.

5.1.1.1 Application Framework Objects

The AF implements an object-oriented language for expressing the design of an application. The key elements of this design are the set of modules and the interconnections between them. Additional objects support runtime data collection and debugging. Thus, modules, connections, logging taps, and breakpoints constitute the Application Framework's primary objects.

5.1.1.1.1 Modules

The AF allows the Application Architect to select a module object for inclusion within an application. This selection process results in an instance of the module being added to an application, thereby allowing multiple instances of a module to exist within a single application. All module objects to be included in an application must have been previously built with the Module Framework and placed in the module catalog. Any documented runtime requirements specified for a module by the Component Embedder are accessible from within the AF. This information helps the Application Architect to specify a default host assignment for each module within an application.

5.1.1.1.2 Connections

The AF allows the Application Architect to specify connections between module objects delineating the flow of application-level (inter-module) messages. A connection is a link from an output port of the module initiating the message as the sender, to an input port of another module acting as the receiver. The message may or may not result in an acknowledgment. The behavior of a connection is determined by the options defined for each port associated with the connection as well as the signatures of those ports.

Port connections must have compatible signatures. Signatures are compatible if the port parameters match in number and datatype for both initial and reply parameters. Corresponding parameters in each port signature may have different names but must be typed using the same datatype.

Output port options define the acknowledge requirements as one of "none," "immediate," or "future." If "none" is specified, the sender passes data to the receiver and does not wait for a reply. Any reply information which may be

generated by the receiver is simply ignored. An "immediate" acknowledge requires the sender to wait indefinitely until the receiver acknowledges the message. Finally, choosing the "future" option allows the sender to perform additional processing before eventually verifying that the receiver has indeed acknowledged the message.

5.1.1.1.3 Logging Taps

Application-level logging taps provide the means to easily (1) globally enable and monitor the same type of taps across multiple modules, and (2) define and enable user-defined groups of logging taps. An Application Architect's desire to log all port activity would be an example of the former capability. In this case, a single, global enabling step would result in all modules having the logging taps associated with each of their ports selected. The latter capability allows the user to group seemingly unrelated logging taps under a single, user-named group to facilitate application-level debugging, monitoring, and analysis. In either case, the enabling of application-level logging taps results in the MCM Workstation individually enabling each affected logging tap on each affected module.

5.1.1.1.4 Breakpoints

Breakpoints are provided at the application level to support the global enabling and monitoring of breakpoints in a manner analogous to that provided for logging taps. Groups of similar breakpoints or user-defined groups of breakpoints may be defined at the MCM Workstation. The enabling of each application-level breakpoint results in the MCM Workstation individually enabling each affected breakpoint on each affected module.

5.1.1.2 Application Framework Editor

The Application Framework provides a graphical editor, referred to as the AF Editor, to support the Application Architect both in defining the connections between modules and in defining application-level logging taps and breakpoints. Creating an application in the AF is accomplished by initially identifying the specific modules to include in the desired application from the catalog of modules previously defined using the Module Framework. The connections between these modules are then specified graphically by selecting pairs of ports to connect. These links are checked for argument and datatype compatibility by the testbed to avoid the generation of illegal configurations.

A menu-oriented approach is used to define application-level breakpoints or logging taps. For example, to define a new group of logging taps, the user is provided with a menu of available modules. Each selected module, in turn, provides a menu of available logging taps within that module for possible inclusion in the new group. Existing user-defined groups may have their members individually deleted using a similar menu-oriented approach.

Finally, the AF Editor allows the Application Architect to invoke other operations on the application as required, such as storing an application definition in the application catalog or retrieving an existing definition from the catalog.

5.1.1.3 Distribution Information

Designing an application architecture requires the assignment of individual modules to specific computing resources within the AAITT. This process is supported by Distribution Information, which includes any Component Embedder-generated comments associated with a module. These comments are used to describe any hardware, language, processing, or resource requirements imposed by the module. The Application Framework provides the Application Architect with a means of reviewing this data. The Application Architect is, in turn, responsible for interpreting these requirements and devising module-to-host assignments which satisfy established goals using available resources.

This Distribution Information is specified using the same menu-based selection of host assignments and status provided for other MCM Workstation actions.

5.1.2 Module Framework

The Module Framework (MF) is the tool used by a Component Embedder to construct new AAITT modules. Figure 5 shows a Module Framework window and a menu of Module Operations. The Embedder starts with the component to be embedded (either actual code or a description of the code) as well as a concept of how that component might be used in an AAITT application and proceeds to construct a graphical module model. The model represents the Component Interface Manager, implementing the runtime interface between the component and the AAITT. Figure 6 shows a Planning Module Model.

5.1.2.1 MF Objects

The MF implements an object-oriented language for expressing the design of a module. The framework separates the external specification of a module into two parts: (1) the manner in which the module interacts with its peers within an application (i.e., the module's application protocol), and (2) the manner in which the module interacts with the DPS as a generic AAITT-compliant module (i.e., its AAITT protocol). The MF is used to graphically define, annotate, and connect various objects allowing the Component Embedder to express both parts of the module's external specification. In addition, the MF provides objects which define the interface to the component, the logging of various runtime data, and the specification of breakpoints.

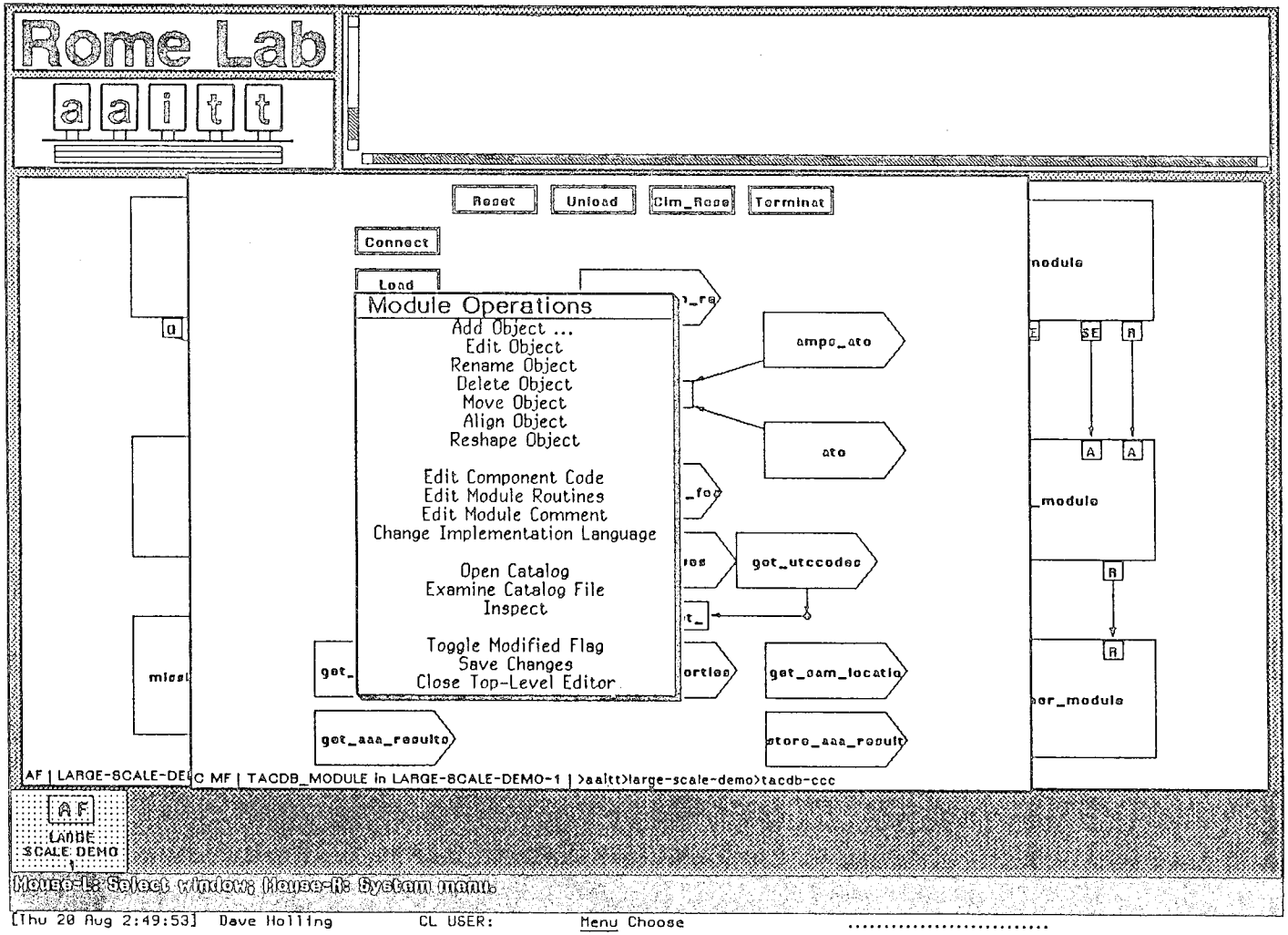


Figure 5. The Module Framework

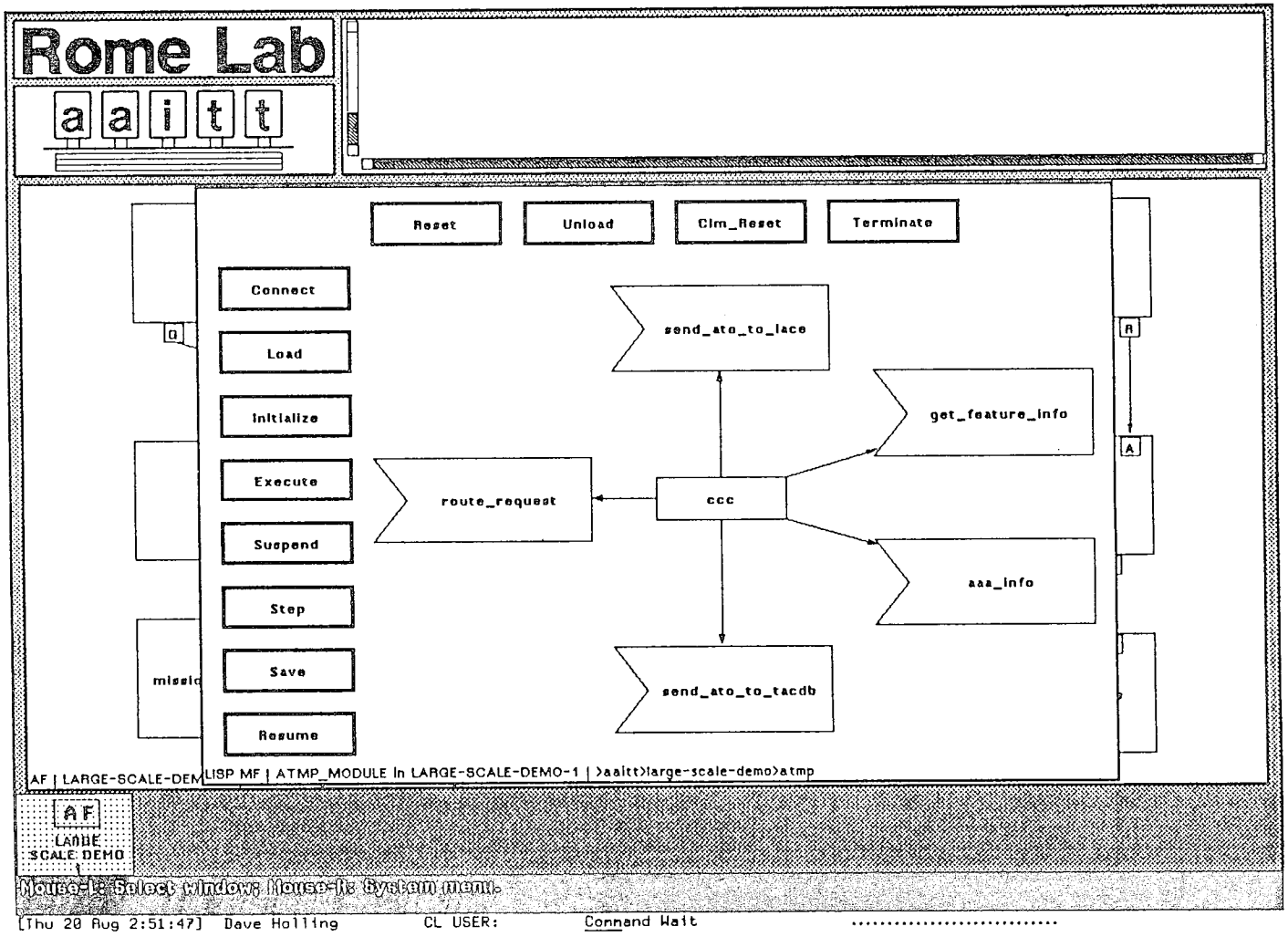


Figure 6. Planning Module Model

5.1.2.1.1 Ports

The MF provides objects called ports as the means to specify a module's application protocol. Each port corresponds to a specific message type that the module can use to communicate with its peers. An MF-resident module initially starts with no ports. The Component Embedder has the responsibility to determine and define the set of ports for the module.

The definition of each port consists of a pair of signatures, a definition of options, and a port body. The port body for an input port is implemented as a procedure which is invoked when data transits the port. The definition of options associated with a port provides the user with a supplementary means of specifying the behavior of a port and results in additional code and/or comments being automatically created for the port body by the testbed's code generators.

5.1.2.1.2 Operations

Operations are used to specify a module's standard set of actions, one for each of the messages defined by the fixed, DPS protocol. As with ports, each operation consists of a pair of signatures and an implementation, which is called an operation body. Each operation has predetermined signatures established as part of the DPS protocol. The MF requires the Component Embedder to extend some of these operations with component-specific actions, and provides the option to augment others.

An operation body defines the means for invoking a DPS operation (e.g., Execute, Suspend, Terminate) on a component. It is implemented as a procedure executed when an operation in invoked by the DPS and depends on the particular implementation parameters of the component. In many cases, the operation bodies automatically generated by the MF can be used with little or no modification. Just as with port bodies, the Component Embedder is responsible for programmatically defining the method for interfacing to the component and for controlling the component. Some components may not easily support the implementation of all DPS operations (e.g., Suspend). It is the responsibility of the Component Embedder to determine whether such an operation should generate a warning message to the calling routine or simply be treated as a "no-op."

5.1.2.1.3 Subroutines

Subroutine objects are a way of organizing code specified by the Component Embedder. As with ports and operations, a subroutine consists of an associated subroutine body. However, unlike ports and operations, subroutines are not part of the module's external interface. Subroutines may be used to encapsulate code which is called by other ports, operations, and subroutines.

Due to the similarity of the code used to implement ports, operations, and subroutines, this user-supplied code is often referred to as a "code body."

5.1.2.1.4 Data Stores

Using the MF, the Component Embedder is capable of defining data stores which aid in the definition of port and operation bodies. Data stores can contain both application- and testbed-level data. Each data store definition consists of a name and cantype. In addition, multi-valued data stores (FIFO, LIFO and Sorted) possess an integer parameter indicating the maximum number of data values which may be stored. The special value "0" for this parameter indicates that the stores can contain an arbitrary number of data items. The priority function for sorted queues is specified dynamically at runtime and is not part of the data store's definition.

Each data store provides a set of functions which can be called from within port, operation, or subroutine code bodies to store and retrieve its data; determine the number of elements it contains; as well as ascertain if it is empty or full.

5.1.2.1.5 Logging Taps

Module-specific logging taps, which record dynamic information about the module at runtime, can be defined by the Component Embedder using the Module Framework. These logging taps are in addition to the default logging taps defined implicitly within port, subroutine, and data store definitions.

Each logging tap definition consists of a name and the signature of runtime data to be recorded. The logging tap definition implicitly defines a logging function which can be called from within a code body. Component Embedders determine the conditions under which a logging tap is called by placing explicit calls to the tap in their code. Logging taps, whether defined explicitly or implicitly, can have their logging activity enabled and disabled dynamically at runtime.

5.1.2.1.6 Breakpoints

Application execution can be suspended using module-specific breakpoints defined using the Module Framework. These breakpoints are in addition to the implicit breakpoints found within port, subroutine, and data store definitions.

A breakpoint definition consists simply of a name and implicitly denotes a break function which can be called from within a code body. Component Embedders specify the breakpoint's triggering conditions by placing explicit calls to it in their code.

5.1.2.1.7 CIM-to-Component-Communication

A generic CIM-to-Component-Communication mechanism is available from the Module Framework for UNIX-hosted components. This socket-based model can be used to exchange messages between CIMs and their corresponding components. Implemented as a subroutine library, the generic CCC interface can be invoked from either the CIM or the component.

5.1.2.2 Module Framework Editor

The Module Framework provides a graphical editor, known as the MF Editor, to support the Component Embedder in the task of either defining new modules or viewing and modifying existing modules. The definition of a module in the MF Framework consists of interconnected graphical icons which correspond to the port, operation, subroutine, data store, logging tap, breakpoint, and CCC objects discussed above.

In general, each different object type has a unique graphical representation. In addition, the links between these graphical objects indicate the underlying relationships between the entities. For example, an arrow from a data store to a port signifies that the port's body reads a value from the data store.

The MF Editor allows the user to modify the module by creating new objects, modifying or deleting existing objects, and connecting objects. The Editor possesses knowledge of the legal graphical syntax of a module model and prevents the Component Embedder from making illegal modifications, e.g., renaming or deleting standard operations defined by the DPS.

Each graphical object placed within the model using the MF Editor results in the corresponding programmatic definition of that object being inserted into the automatically-generated code frame, or skeleton, for that module. The programmatic definition includes the specification of any access or monitoring functions required to interface code bodies with the object.

Finally, the MF Editor allows the Component Embedder to both store a completed module definition in the module catalog as well as retrieve an existing definition from the catalog.

5.1.3 Datatype Framework

The Datatype Framework is a forms-based tool used to extend the AAITT's basic set of datatypes, or cantypes, as well as define signatures in terms of both required and optional sets of cantypes. Cantypes and signatures are saved in the catalog for later use by the Component Embedder.

5.1.3.1 Cantypes

The testbed provides thirteen primitive cantypes, as shown in Table 2. These cantypes include integers, booleans, strings, and arrays, as well as Cronus-specific types that are used internally by the DPS. This basic set of cantypes can be extended by defining enumerations and structures.

Datatype Name	Description
U16I	Unsigned 16-bit integer
S16I	Signed 16-bit integer
U32I	Unsigned 32-bit integer
S32I	Signed 32-bit integer
F32	Floating point single
F64	Floating point double
ENUM	Enumerated type
EBOOL	Boolean
ASC	ASCII string
ARRAY	Array of some simple type
EDATE	Timestamp
EINTERVAL	Time difference
UNDEF	Application private type

Table 2. AAITT Canonical Types

5.1.3.2 Signatures

Signatures define the information which is passed between modules as well as the information passed to objects inside the module. A signature is a collection of named arguments. Each argument is either required or optional. It should be noted that each port and internal object actually require two signatures. The first defines the data passed in, while the second defines the data returned.

5.1.4 Catalog System

The MCM Workstation provides a Catalog which includes applications defined with the Application Framework, modules created with the Module Framework, as well as cantypes and signatures defined using the Datatype Framework.

5.1.4.1 Structure

The Catalog is an abstract representation of a file system directory, allowing the Component Embedder or Application Architect to organize AAITT entities without having to be familiar with the underlying operating system file structure by managing all of the required details. The catalog hierarchy corresponds precisely to the underlying directory structure. A catalog specifies the directory in the file system that corresponds to the root of a catalog hierarchy. Thus, each catalog corresponds to a directory "tree" in the file system.

The catalog supports a user-defined hierarchy of files to provide the Application Architect with the freedom to separate stable configurations from experimental or developmental configurations. It is the responsibility of the Application Architect to define, maintain, and organize this catalog to support the needs of all AAITT users.

5.1.4.2 Version Control

Version control is offered for the Catalog. However, the versioning mechanism provided by the file system of the computer used to host the MCM Workstation directly determines the extent of this capability.

5.2 Control

Control Tools allow the Application Architect or Application User to control and interrogate AAITT applications. The selection of a previously-defined AAITT application causes the application's graphical representation to be displayed. This graphic display becomes the control interface for the application at the MCM. The interactive control interface serves the dual purpose of displaying application status as well as providing the means to control either the entire application or individual modules.

5.2.1 Status Display

The MCM Workstation's Control Tools provide the Application Architect or Application User with a continuously-updated graphical display of module status for the selected application. Figure 7 shows an MCM Control Tools display along with the MCM Control Tools menu. The status of each module within the application as well as the connections between modules is indicated graphically as described in Table 3.

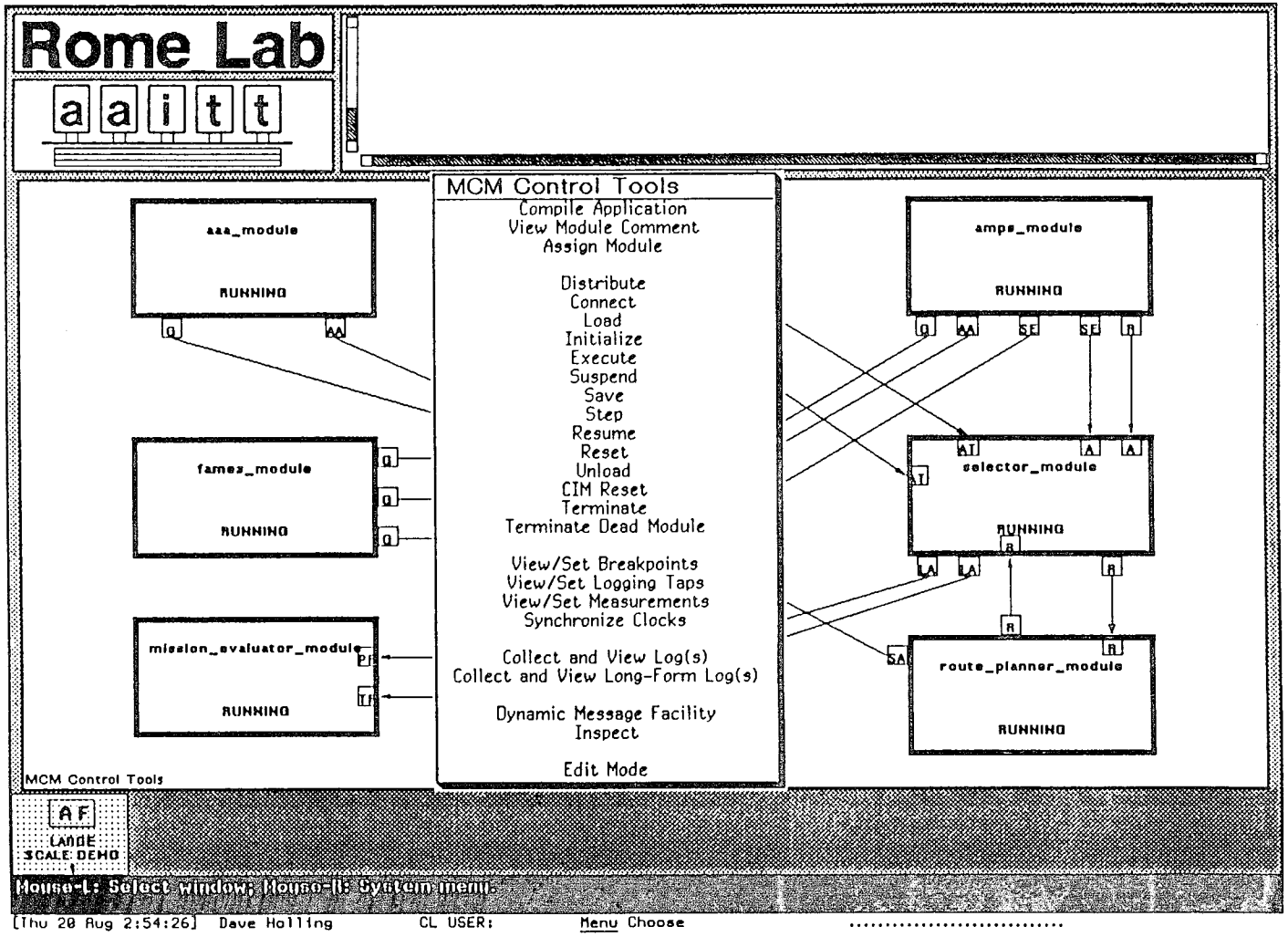


Figure 7. MCM Control Tools

Status	Corresponding Display
Application Selected	The model of the application is displayed with low-contrast or dashed borders and labels. Connections are also shown with the same low-contrast representation.
CIM Loaded	A module is said to have its CIM Loaded once the CIM has been distributed to the assigned host and the assigned host is executing the CIM. This state is represented graphically with a solid border for the module icon.
CIM Connected	A module is in the CIM Connected state after performing any initialization steps required to establish connectivity between that CIM's output ports and any other CIMs in the application. This state is represented by showing the relevant connection as a solid line.
Loaded	A module has been loaded when the CIM has loaded the associated component on the assigned host computer. A loaded module is represented graphically by a module icon with a dark/solid border and dark/solid text.
Initialized	An initialized module is represented graphically with dark/solid/thick borders and text and a low-contrast background.
Running	An executing module (CIM and component) is indicated by highlighting the entire graphic icon for the module.
Paused	A paused module is shown by altering either the module's border or entire icon.

Table 3. Status Display Descriptions

5.2.2 Compile / Assign Modules

The executable CIM for each module within an application is generated as a result of compiling the application model using the Control Tools. Figure 8 shows the MCM Control Tools menu used to compile applications as well as a Host Assignment window for assigning modules to host machines.

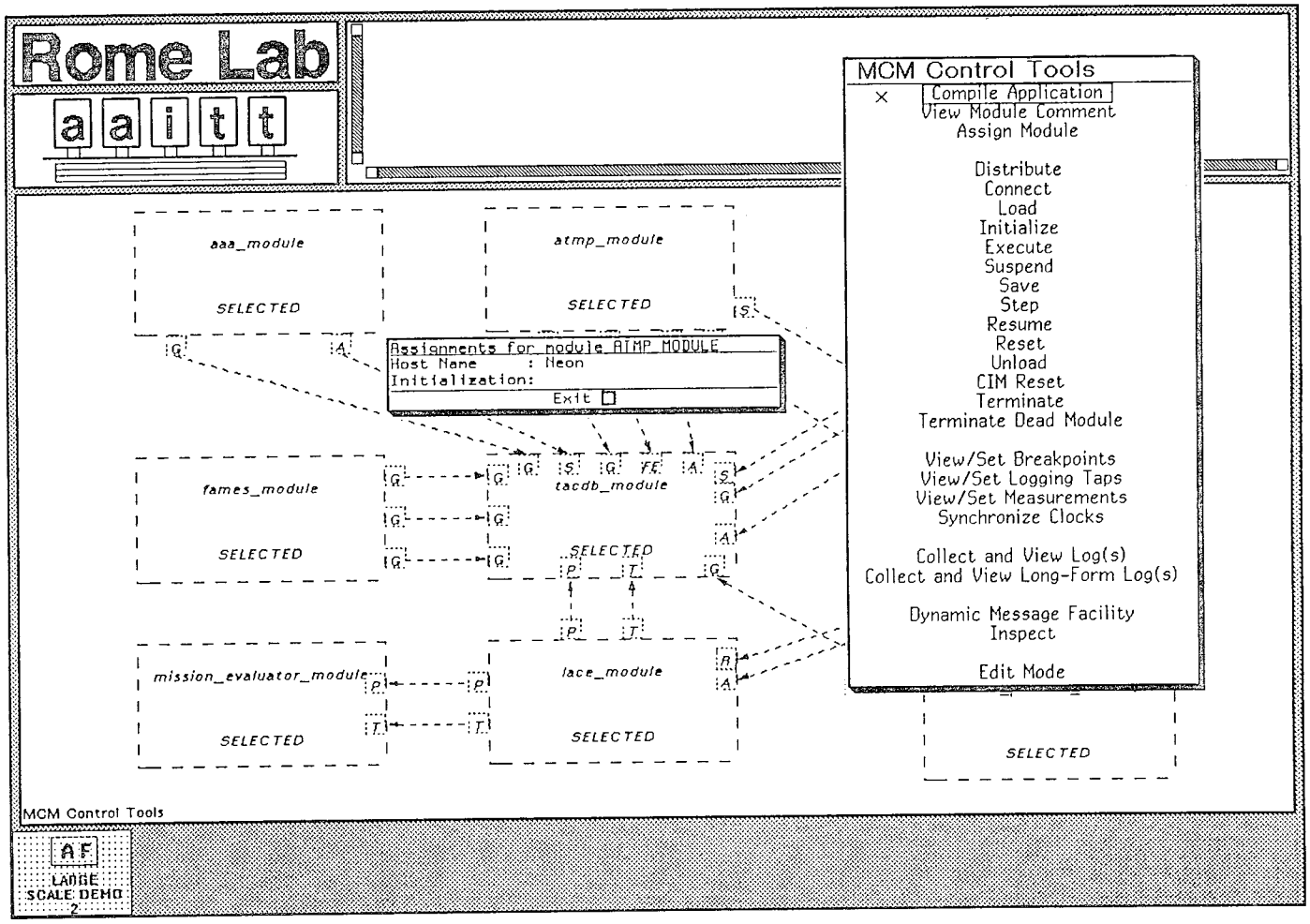


Figure 8. Application Compilation and Host Assignment

The AAITT compilation process requires three items: the definitions of each module in the application; the datatype definitions referenced by each module; and the links defined for the application.

An application model may be developed entirely at the MCM Workstation, without involving the specific host computers associated with each module in the application. However, compiling the application model to generate CIMs for each module requires access to some or all of the hosts due to local compilation activity.

The compilation process uses the application and module models, along with any code annotations, to produce a set of AAITT modules which can be executed. The AAITT compiler first produces a set of source files corresponding to each module in the application. It then invokes the appropriate language compilers as well as manages the process of linking the resulting object files to the appropriate AAITT and language libraries to produce executable files. This process is initiated and controlled entirely by the MCM. No user intervention is required, even in cases where the compilation process is occurring on several different host machines.

As with any programming activity, the user's code may contain errors which become apparent at either compile or runtime. The MCM displays any warnings or errors detected at application compilation time. Similarly, the application's design may be incomplete or incorrect, necessitating changes in the application architecture. The compiler records all relevant aspects of the application being compiled to support incremental compilation. Thus, when the application's structure or code is changed and the application is recompiled, the compiler intelligently performs the minimal amount of work necessary to properly reflect changes within the new executable code.

After compilation, each module is assigned to a host machine. A CIM can be distributed to any machine that supports the CIM's full computing environment, including any supporting software, such as an expert system shell, a GUI (Graphical User Interface) package, and so on. Therefore, a CIM compiled on a Sun computer can be assigned to any compatible Sun machine supporting the CIM's configuration.

5.2.3 State Transition

Once an application has been compiled, the MCM Workstation's Control Tools are used to transition modules through their available states. The set of states defined for a module are {CIM LOADED, CIM CONNECTED, LOADED, INITIALIZED, RUNNING, PAUSED}. Note that these states apply to each individual module within an application. All modules within the application are not required to be in the same state.

The first two of the states, CIM LOADED and CIM CONNECTED, reflect a change in the process state of the CIM. The remaining CIM states, LOADED, INITIALIZED, RUNNING, and PAUSED, all reflect a change in the process state of the component associated with the CIM.

Unlike all other state transition messages, the DPS command "Distribute" is not sent to a module but, instead, is implemented directly by the Distributed Processing Substrate. The command creates a CIM on the appropriate host and places the CIM in the CIM LOADED state. Once a CIM has been placed in this state, the CIM's full functionality is enabled and it may be controlled using the AAITT's DPS protocol.

The AAITT's state transition commands are presented below and include an explanation of the activity initiated by each command.

5.2.3.1 Distribute

The "Distribute" command instantiates the CIM associated with the selected module on the appropriate host processor. If there is no host assignment, the MCM host processor is used.

5.2.3.2 Connect

The "Connect" command establishes the connections associated with the input and output ports of the selected module. The links between modules, in effect, become instantiated with the DPS-level address(es) needed by that link to pass or receive data. The connections are then tested to ensure that communication can occur.

5.2.3.3 Load

The "Load" command instructs a CIM to load its associated component on the local computer. This capability is particularly useful in persistent environments, such as on Symbolics machines, where applications must be loaded into the current image.

5.2.3.4 Initialize

Modules may require various forms of initialization activity to occur prior to execution. The "Initialize" command was specifically provided to support these type of operations. Examples of initialization activity include establishing the module's configuration, the definition of a goal or context, or access to a startup file. This command is also intended to support the reinstantiation of any previously saved module instances.

5.2.3.5 Execute

The "Execute" command initiates execution of the component associated with a CIM. This command's activity is frequently simplified because the "Initialize" command is often used to resolve the component's startup and/or configuration issues.

5.2.3.6 Suspend

The "Suspend" command allows the Application Architect or the Application User to stop module processing at any time. However, it is important to note that the effect of this operation depends on the selected component. Some components may not be easily suspended in a manner which allows them to cleanly resume operation at a later time. Other components may readily support this operation. It is the responsibility of the Component Embedder to either support this capability or note its absence.

5.2.3.7 Resume

The "Resume" command is used to restart a module previously suspended by a breakpoint or by the "Suspend" command. The effect of this operation depends on the component being resumed. Some components may not support the resumption of execution once they have been suspended.

5.2.3.8 Terminate

The "Terminate" command halts the selected module's component and terminates its CIM. Once terminated, a module must be restarted beginning with the "Distribute" and "Load" command sequence.

5.2.3.9 Reset

The "Reset" command suspends the associated component and resets a module to its Loaded state. The module must be subsequently Initialized before it may be Executed again.

5.2.3.10 Unload

The "Unload" command results in the termination of the associated component. However, the module's CIM remains loaded and all connections remain intact. This command places a module in the CIM Connected state.

5.2.3.11 CIM Reset

The "CIM Reset" command results in the termination of the associated component. In addition, all connections associated with the CIM are reset. The

CIM continues executing on its assigned host. This command places the module in the CIM Loaded state.

5.2.4 Breakpoint Control

The MCM Workstation's Control Tools provide a means for setting, detecting and reviewing breakpoints within an application. The setting of a breakpoint at the MCM Workstation results in a flag being set within the appropriate CIM. The CIM then checks this breakpoint flag during the normal execution of the code body associated with the particular CIM port or operation selected as a breakpoint. The setting or resetting of a breakpoint is performed at the MCM Workstation by selecting the module of interest and choosing the breakpoint command using the mouse. This results in the display of a menu showing all of the breakpoints associated with the module, including those defined by the Component Embedder using the Module Framework, as well as the set of all default breakpoints associated with each port or operation.

The AAITT automatically defines several types of breakpoints as a result of the module- or application-definition process. These breakpoints as referred to as built-in breakpoints. The testbed also supports the definition of additional breakpoints to augment the built-in breakpoints. These breakpoints are referred to as user-defined breakpoints.

5.2.4.1 Built-In Breakpoints

The AAITT provides several types of built-in breakpoints to monitor DPS-level, module-level, and application-level events within the testbed.

DPS-level events include all operation activity. An operation event is triggered when the appropriate DPS message is received or sent by the module.

Module-level events include all port activity and data store access. A port event is triggered when data passes through that port. A data store event is defined as reading/writing from/to a data store.

Application-level events provide an easy means of globally enabling and monitoring the same type of breakpoints across multiple modules. The enabling of an application-level breakpoint results in the MCM Workstation's Control Tools individually enabling each affected breakpoint within each module. Several application-level breakpoints of this type are provided as part of the Control Tools, including: (1) All Port Activity Within the Application, (2) All Operation Activity Within the Application, (3) All Operation Activity Of <a particular type> Within the Application, and (4) All DPS-Level Events Of <a particular type> Within the Application.

5.2.4.2 User-Defined Breakpoints

The Component Embedder is provided with the means to specify user-defined breakpoints, either within the CIM or within the component itself. A breakpoint within the CIM can be defined within a port, operation, or subroutine body, i.e., a code body, by the Component Embedder. A breakpoint of this type will typically be triggered by a particular pattern of messages, arbitrary variable values, or combinations of these and other predicates. The Component Embedder may also modify the component itself to signal an event that would otherwise not be available to the CIM. The CIM could, in turn, trigger the breakpoint based either simply on the new event or in combination with any other activity within the CIM.

The Application User is also able to specify user-defined breakpoints as well as group seemingly unrelated breakpoints into a user-named group to facilitate application-level debugging, monitoring and analysis. The Control Tools' menu-oriented interface provides the user with a mousable list of breakpoints for inclusion in a new breakpoint. Breakpoints defined in this manner can also be edited, with new breakpoints added or existing breakpoints deleted.

5.3 Monitoring

The MCM Workstation's Monitoring Tools provide the means to review and interrogate an application without pausing or halting normal processing. The Monitoring Tools allow the Application Architect or Application User to turn logging taps on/off, collect logging tap data, analyze and filter logging tap data, as well as review these results in a tabular or graphic fashion. The Monitoring Tools also permit the dynamic query of status within each CIM.

5.3.1 Logging and Analysis

The AAITT provides a logging capability to capture dynamic information at runtime for later analysis and presentation to the Application Architect. A logging tap is a piece of code which, when enabled, stores a datum known as a log entry in a log database. Typically, the log entry datum is a structured object which contains several fields. Each field contains a subdatum, or element, describing the particular circumstances surrounding the invocation of the logging tap, e.g., the time the tap was called, the task that contained the tap, variable values, etc. The placement of logging taps and the data they record is determined by the kinds of analyses in which the Application Architect is interested. Generally, taps are used to record activity within modules and the flow of messages between modules.

The Application Architect or Application User identifies logging taps in the same manner that breakpoints are specified using the Application Framework. Figure 9 shows the View/Set Logging Taps selection menu.

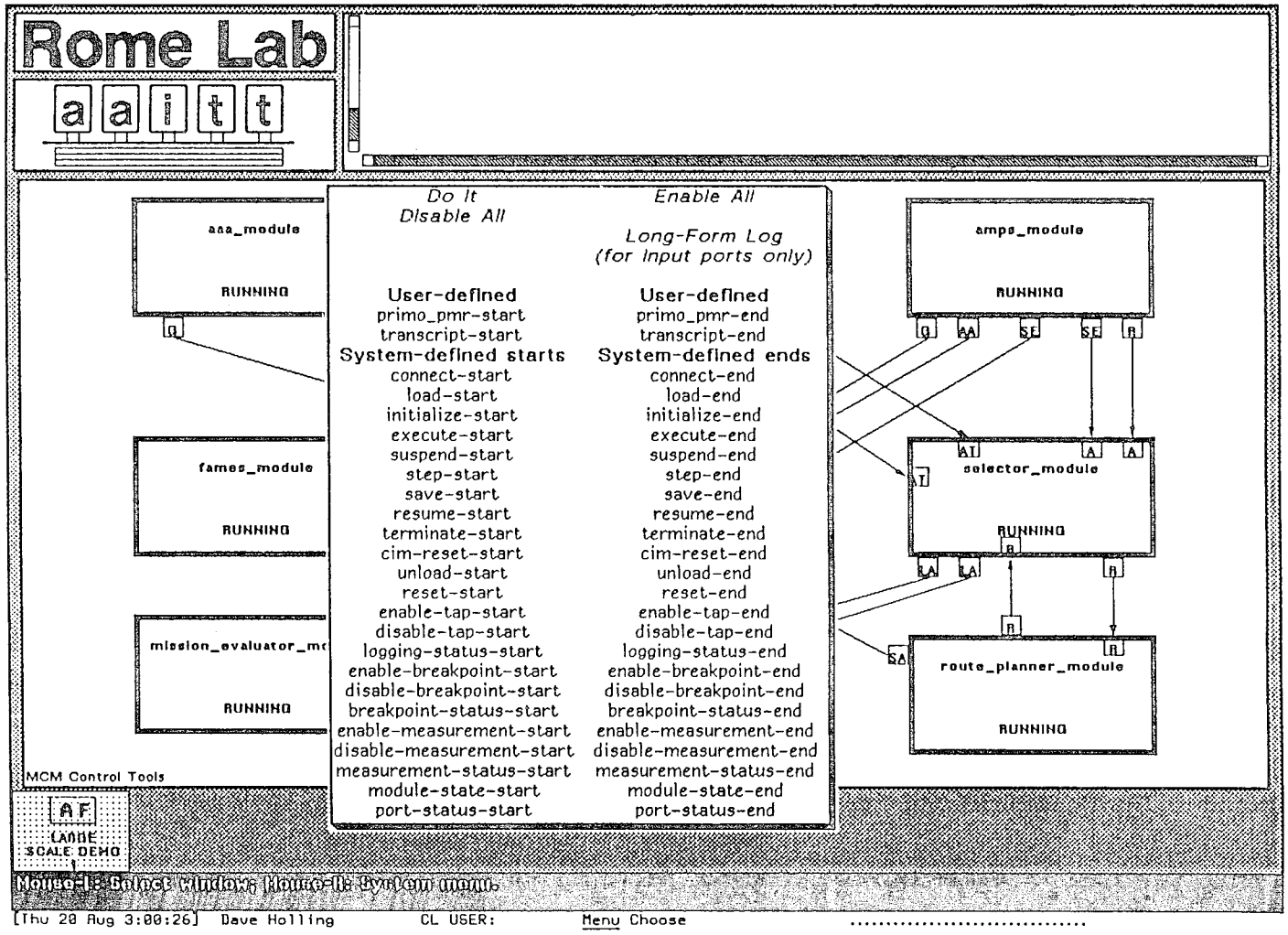


Figure 9. Viewing and Setting Logging Taps

Several types of logging taps are automatically defined as a result of creating a module or an application. These logging taps as referred to as built-in logging taps. The testbed also supports the definition of user-defined logging taps to augment the built-in taps. Both types are discussed next.

5.3.1.1 Built-In Logging Taps

The built-in logging taps provided by the AAITT monitor DPS-level, module-level, and application-level events. These are the same events which trigger the built-in breakpoints described above. In addition to regular logging, the testbed offers the ability to capture the entire contents of messages sent to and replied from input ports. These long-form log messages are available to AAITT users in a human-readable form to facilitate debugging. This data can also be saved within a file and edited for use as input to the AAITT's Dynamic Message Facility, discussed below.

5.3.1.2 User-Defined Logging Taps

The support for user-defined logging taps within the testbed is similar to that which is provided for user-defined breakpoints. The Component Embedder may add logging taps to the CIM's code bodies or modify the component itself to signal an event that would otherwise not be available to the CIM. The Application User is also able to define logging taps by grouping seemingly unrelated logging taps into a user-named group to facilitate application-level debugging, monitoring, and analysis.

5.3.1.3 Logging Tap Control

When a standard or long-form logging tap is enabled at the MCM Workstation, a flag is set within the appropriate CIM. This logging flag is checked as part of the normal processing of the code body associated with the particular CIM port, operation, or subroutine selected for logging. The enabling or disabling of a logging tap is performed at the MCM Workstation by using the mouse to select the module of interest and then choosing the Logging command. A menu of the operations which may be performed on the module's logging taps is presented to the user.

Both built-in and user-defined logging taps are handled in the same manner by the testbed. The logging operations which may be performed at the MCM Workstation include: (1) Logging Control, (2) Collection, (3) Analysis and Filtering, (4) Presentation, (5) Save, as well as (6) Restore. These operations may be performed while the application is running, or after the application has been paused due to the triggering of a breakpoint or the invocation of the "Suspend" command. Some operations, such as Collection, may cause elapsed application execution times to increase due to the potentially large data transfers required of the DPS. This overhead is not reflected in the log data. In this way, logs truly represent what they were intended to measure.

Logging taps may also be dynamically enabled or disabled at the MCM. When an application is executed, all enabled logging taps will insert entries into the AAITT log file maintained on each local host. The MCM Workstation provides for the subsequent collection and transfer of these logs from the various hosts within the AAITT to the MCM. Typically, log collection and transfer is requested when a breakpoint has been reached or upon the completion of an application run. The MCM is also capable of gathering logs during application execution. However, this activity may adversely affect timing.

5.3.1.4 Filters

Once the desired logs have been transferred to the MCM Workstation, additional AAITT tools are provided for combining, filtering, and analyzing these logs. These tools are collectively referred to as filters. Examples of filters which might be found at the MCM Workstation include:

- Merge Logs
- Time Extraction/Exclusion
- Event Type Extraction/Exclusion
- Relationship or Pattern Recognition
- Data Transformation Algorithms

The MCM offers the ability to locally store and retrieve both raw and filtered log files, thereby eliminating the need for users to repetitively perform basic filtering tasks in cases where they are interested in performing multiple analyses of the same data. All log data, whether raw or filtered, may be viewed at the MCM Workstation using either a graphical display for time-dependent information or a tabular display.

5.3.2 Debugging

Logging taps and breakpoints provide both Application Architects and Application Users with a robust means of monitoring, controlling and analyzing an application. Thus, they act as the primary tools for debugging AAITT applications and are meant to supplement each individual component's logging and breakpoint capabilities. The analysis of breakpoint and logging tap data as described above is ideally suited for the debugging of an application or module that is not executing as expected or desired. The same techniques are also valuable for assessing various architectures by quantifying application-level processing efficiency measurements or processing results. Figure 10 shows the AAITT Metrics Analyzer, which is used to display events recorded during application execution. It is used extensively to help resolve timing problems and debug new module connections.

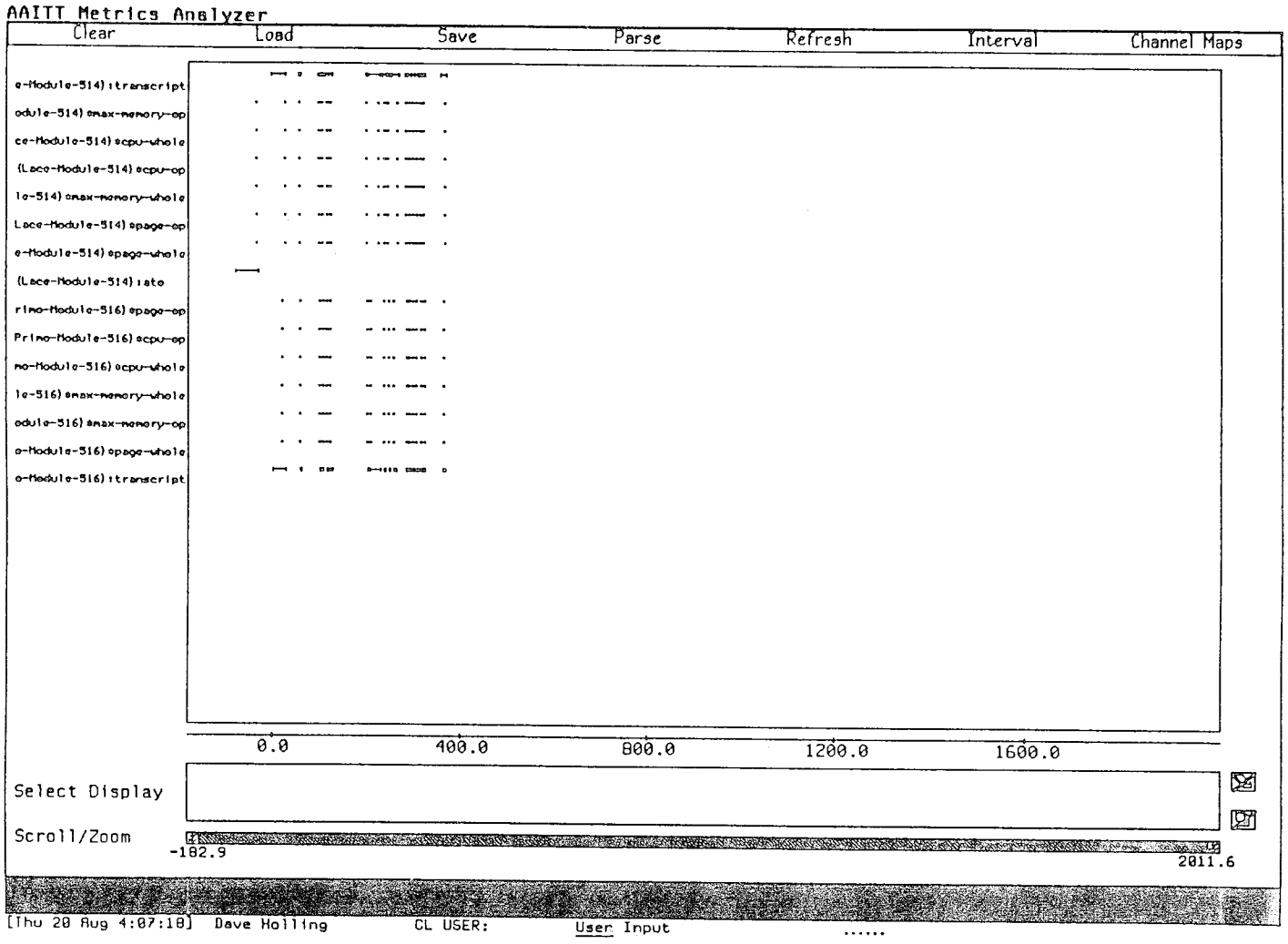


Figure 10. AAITT Metrics Analyzer

5.3.3 Dynamic Message Facility

The Dynamic Message Facility supports the user's ability to dynamically generate and send application protocol messages to a receiving CIM, enabling the progressive, interactive testing of a particular application message interface. Thus, the facility can be used to test the interface of a particular port within a partially or fully-completed 'receiver module' without the need for developing the 'sending module,' as long as the relevant port signature(s) and interface were previously defined using the Module Framework. In addition, the receiving module and its CIM must be in the execute state to run the facility.

The Dynamic Message Facility's human-computer interface is menu-driven, allowing the user to easily select a message-generation mechanism, specify the message's contents, and identify the receiver module. The interface module collects the user's typed entries as well as menu selections, and uses this data as parameters to remotely invoked UNIX shell scripts and their supporting binary programs.

Based on the input parameters received, the shell scripts either generate a new message template or parse an existing message template or long-form log output. A file of data suitable for use as input to the Cronus *tropic* tool is then dynamically constructed before *tropic* is invoked to send the dynamic message.

Any results from the message invocation are stored in a file. If the appropriate menu selection has been made, the information returned from a module that received a request for data via a dynamic message will be displayed in a pop-up editor window. Dynamic messages which are not data requests will result in the name of the port being displayed followed by a colon. An error message will be returned if the message was not successfully delivered or processed.

Although the Dynamic Message Facility ultimately relies on the invocation of *tropic*, the facility represents an improved testing capability for message interfaces which is an integral part of the AAITT's MCM Workstation. In addition, the facility presents a uniform interface already familiar to the AAITT Component Embedder through the use of entities such as signatures, ports, and modules.

5.3.4 Measurements

The AAITT also defines automatic logging taps which record various low-level measurements of system resource usage, such as CPU, memory, etc. These performance logs, called 'measurements' may be enabled and disabled on a per-resource basis. That is, if the user turns on the "CPU-Usage" measurement, then CPU-usage will be recorded at the occurrence of every DPS event in the module. Measurements are dynamically enabled or disabled at the MCM Workstation using the View/Set Measurements menu, as shown in Figure 11.

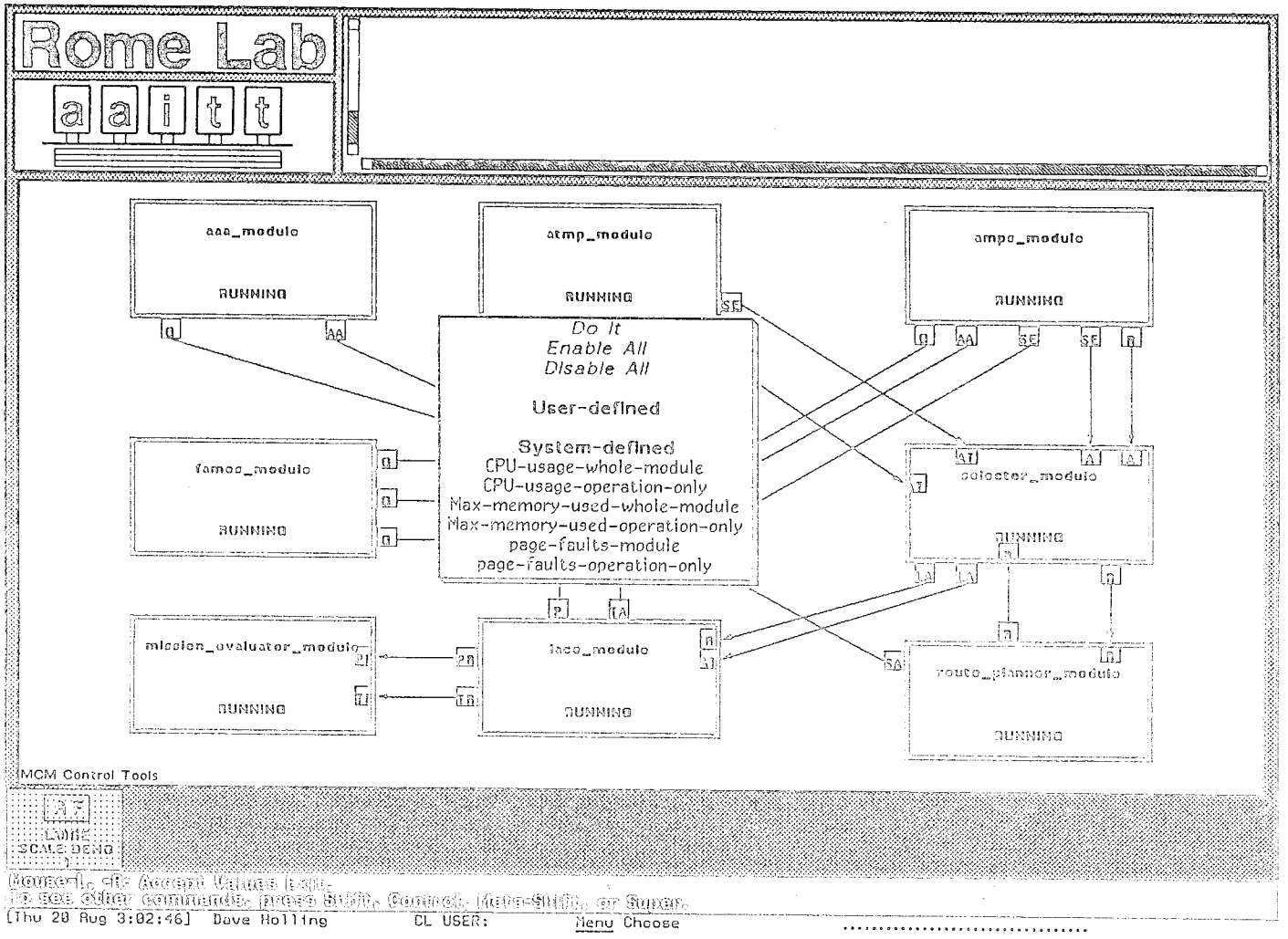


Figure 11. Viewing and Setting Measurements

The particular resources which may be measured depends upon the type of support provided by each host's operating system. Measurements are collected and otherwise treated as ordinary log events.

5.4 Synopsis

The AAITT's development paradigm utilizes iterative modeling, control, and monitoring activities. Users are asked to (1) configure application suites graphically to effect encapsulation, (2) measure application and component behavior during execution, and (3) analyze information about key events. This cycle is repeated to investigate alternate solution strategies.

The testbed's toolkit embodies advances in modeling, code generation, control, and the use of performance metrics to raise distributed system development to new levels.

Modeling Frameworks allow users to graphically introduce desired intercomponent communication and data processing approaches into the testbed. Distributed systems are built via Module-Oriented Programming. Competing architectures can be pictorially expressed and investigated. Crucial data flow and control issues are identified prior to full implementation.

State-of-the-art code generators subsequently transform the models into executable wrappers permitting components to become embedded within the AAITT. Both C- and LISP-language implementations are supported. Incremental compilation dramatically diminishes development cycles. Automated mechanisms manage the complex compilation process.

Control tools minimize the possibility of 'run away' applications. Distributed systems are brought up and down in a controlled fashion. Both conventional and knowledge-based software is accommodated. Breakpoints can be triggered by user-defined conditions or standard events. The ability to single-step applications at the message-level aids debugging.

Finally, measurement, instrumentation, and monitoring capabilities facilitate iterative application development and performance tuning. Selectively-enabled built-in and user-defined measurements drive analyses. Non-intrusive application monitoring minimally impacts performance. An integral Metrics Analyzer displays all time-dependent information.

The resultant AAITT permits testbed users to reap the benefits of applying a comprehensive toolkit within a structured development paradigm.

6 AAITT Applications

The base AAITT program required three specific demonstrations of the testbed's capabilities "in the context of the USAF C2I problem domain." The defined purpose of each was as follows:

Preliminary Demonstration	"to demonstrate the testbed facilities necessary to integrate a minimum set of separate knowledge-based systems sufficient for a serious application demonstration."
Large Scale Demonstration	"to demonstrate the testbed facilities necessary to integrate a full-scale application suite of knowledge-based and conventional systems."
Reusability Demonstration	"to demonstrate that the testbed concept and architecture was applicable to multiple domains."

Guidance received from Rome Laboratory during the program's kick-off meeting resulted in the implementation of a plan that emphasized the use of existing conventional and knowledge-based components for all demonstrations. The motivation for this prudent strategy was to maximize the resources applied to testbed development and minimize expenditures on the applications employed within the demonstrations. An additional benefit of this approach was that each resulting application would consist of mature components. As a result, the team's originally-proposed scenarios for the demonstrations were adapted to fully implement this plan.

Each of the three demonstrations are presented below.

6.1 Preliminary Demonstration

The Preliminary Demonstration integrated the following three Government-furnished components to show the interoperability of existing, independently-developed planning, database, and simulation capabilities, respectively:

- AMPS — Air Force Tactical Mission Planning System
- TAC-DB — Tactical Red and Blue Force Database
- LACE — Land Air Combat in ERIC

More specifically, the Preliminary Demonstration's application utilized AMPS as a mission planner which was responsible for generating an Air Tasking Order

(ATO); TAC-DB for managing all of the operational data required for mission planning and execution within the European theater; and LACE to simulate execution of the AMPS-generated ATO. Figure 12 shows the application architecture for the Preliminary Demonstration as well as the AMPS module model. The demonstration successfully illustrated the testbed's ability to support the embedding and integration of the disparate USAF components into a cohesive problem-solving suite.

The following types of mission simulations were supported by LACE and executed using the Preliminary Demonstration application:

- Offensive Counter Air (OCA)
- Surface-to-Air Missile Suppression (SAM)
- Air-to-Air Refueling (AAR)

Additionally, as presented in Figure 13, the Demonstration displayed the user-oriented, on-line, graphical presentation of system activity which had been specifically included in the AAITT. The testbed's ability to perform monitoring and measurement actions during application execution was also demonstrated. Figure 14 shows an example subset of the raw measurement data captured by the testbed.

A list of suggested additions, changes, and improvements to the testbed was generated as a result of the Preliminary Demonstration. These items can be found in Table 4, along with the specific actions taken by the team to address each suggestion.

6.2 Large Scale Demonstration

To meet the requirements of the Large Scale Demonstration, the team had to establish a coherent problem-solving suite within the domain of USAF Tactical C2I by using the testbed to embed and integrate a full-scale application of knowledge-based and conventional components. The identified problem set would also reuse the AMPS, TAC-DB, and LACE modules and be augmented with at least two more existing components. Additionally, the completed application had to:

- be multi-agent, time sensitive, and factorable into sub-problems;
- possess spatial and temporal aspects, requiring the use of database and planning capabilities, as well as reasoning under uncertainty; and
- be valid with respect to current or proposed concepts of operation, as well as demonstrate the use of the testbed as a training tool and a planning/decision support vehicle.

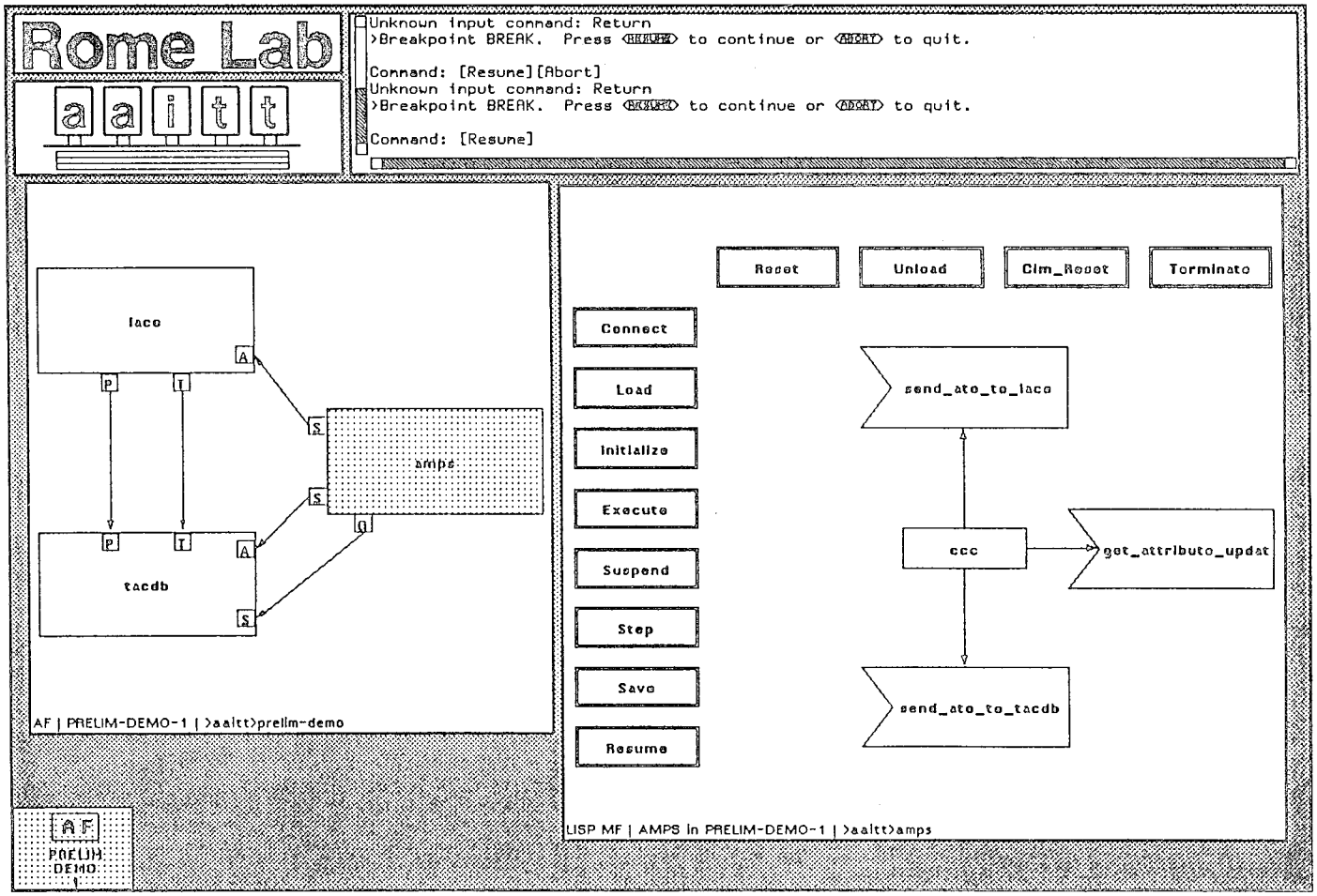


Figure 12. Preliminary Demonstration's Application Architecture and AMPS Module Model

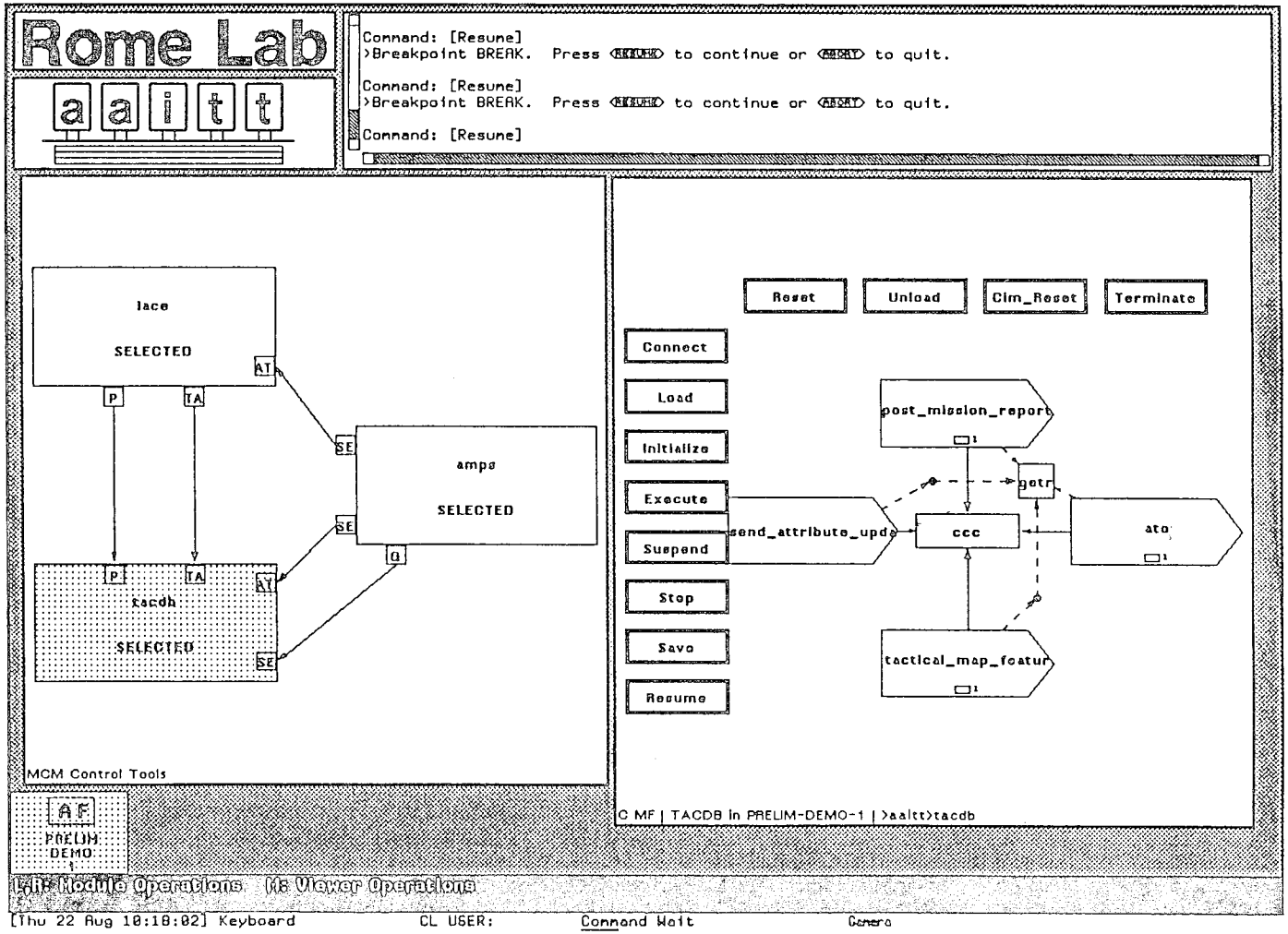


Figure 13. Preliminary Demonstration's Application Framework and TAC-DB's Module Framework

```
(682912060 "getrusage_tap" 1 "{tacdb}" T NIL 7 ("ato" "770000" "60000"))
(682912281 "getrusage_tap" 1 "{tacdb}" T NIL 0 ("send_attribute_updates" "690000" "900000"))
(682912288 "getrusage_tap" 1 "{tacdb}" T NIL 4 ("ato" "730000" "970000"))
(682912380 "getrusage_tap" 1 "{tacdb}" T NIL 9 ("ato" "790000" "130000"))
(682912509 "getrusage_tap" 1 "{tacdb}" T NIL 5 ("ato" "740000" "990000"))
(682912606 "getrusage_tap" 1 "{tacdb}" T NIL 3 ("ato" "720000" "960000"))
(682912642 "getrusage_tap" 1 "{tacdb}" T NIL 6 ("ato" "760000" "30000"))
(682912811 "getrusage_tap" 1 "{tacdb}" T NIL 1 ("send_attribute_updates" "700000" "930000"))
(682912895 "getrusage_tap" 1 "{tacdb}" T NIL 2 ("ato" "700000" "960000"))
(682912912 "getrusage_tap" 1 "{tacdb}" T NIL 8 ("ato" "770000" "110000"))
(682913328 "getrusage_tap" 1 "{tacdb}" T NIL 11 ("ato" "800000" "200000"))
(682913899 "getrusage_tap" 1 "{tacdb}" T NIL 10 ("ato" "790000" "180000"))
(682916005 "getrusage_tap" 1 "{tacdb}" T NIL 16 ("send_attribute_updates" "320000" "680000"))
(682916054 "getrusage_tap" 1 "{tacdb}" T NIL 18 ("ato" "360000" "710000"))
(682916156 "getrusage_tap" 1 "{tacdb}" T NIL 14 ("send_attribute_updates" "310000" "610000"))
(682916258 "getrusage_tap" 1 "{tacdb}" T NIL 13 ("tactical_map_features" "180000" "440000"))
(682916273 "getrusage_tap" 1 "{tacdb}" T NIL 17 ("send_attribute_updates" "350000" "700000"))
(682916622 "getrusage_tap" 1 "{tacdb}" T NIL 15 ("send_attribute_updates" "320000" "630000"))
(682916658 "getrusage_tap" 1 "{tacdb}" T NIL 19 ("ato" "370000" "710000"))
(682916863 "getrusage_tap" 1 "{tacdb}" T NIL 12 ("tactical_map_features" "250000" "80000"))
```

Figure 14. Raw Measurement Data from Preliminary Demonstration

Suggested Addition, Change, or Improvement	Comment / Action Taken
Investigate transitioning the MCM to a SUN workstation.	Implemented and shown at the Reusability Demonstration.
Investigate testbed support for Ada and an upgrade to Cronus 2.0.	Use of Ada-based components not precluded in AAITT. Ada modules can use UNIX-based CCC library. Upgrade to Cronus 2.0 completed by the Reusability Demonstration.
The interface between a CIM and its Component should be generated "automatically."	A UNIX CCC library was implemented prior to the Reusability Demonstration.
The testbed should permit concurrent activity. For example, users should be able to conduct several mission planning tasks simultaneously using multiple threads of control.	This capability is implicit in the design of the AAITT. Concurrent activity is achievable if tasks can either be distributed to different machines or be performed on a single machine which supports multiprocessing.
An applications "switch" is needed.	A switch module was included in the Large-Scale Demonstration.
Testbed users should be able to plug/unplug both measurement and control.	Measurement and control activity is enabled/disabled via point-and-click menus.
The ability to stop simulations, as well as rewind and replay them should be incorporated.	Single-stepping at the message-level was available by the Large-Scale Demonstration. Rewinding and replaying simulations is totally dependent on the simulator's capabilities.
Rome Laboratory personnel should receive a training course covering AAITT use and operation.	A week-long training course was prepared and conducted prior to the Reusability Demonstration.

Table 4. Suggested Testbed Development and Actions

Using Tactical C2I domain expertise present on the team, a fictitious operational scenario was developed for the Large Scale Demonstration. The scenario capitalized on the existence of the TAC-DB unclassified database and can be summarized as follows:

- A European Theater of Operations, including what was once East Germany and the southern region of the former West Germany, was employed to remain within the boundaries of LACE's existing map facilities.

- Politico/ethnic divisions between neighboring factions triggered hostilities between the US-aligned Blueland and aggressors operating from within Redland.

- Redland inflicted significant military and civilian casualties on Blueland using their extensive Close Air Support assets.

- Blueland's premier appealed directly to the President of the United States for assistance in intercepting Redland's air assets and denying Redland operational autonomy by cratering the runways of airbases from which its acts of aggression were being initiated.

- The President issued a directive to the Chairman, Joint Chiefs of Staff, to conduct a military air campaign within the Blueland/Redland Theater of Operations with an overall objective of terminating Redland's hostile air operations.

A comprehensive search was then conducted to identify, evaluate, and acquire additional components relevant to the Tactical C2I domain in keeping with the desire to use existing, unclassified components to assemble the Large Scale Demonstration's application. This search was exhaustive throughout the USAF. Other locations within the other services as well as several offices within the OSD community anticipated to possess relevant components were also canvassed. Generally, small "pockets" of AI use were located throughout DoD. Most of the identified components were not suitable for use because they were designed to satisfy the very narrow and specialized mission requirements of individual systems.

The testbed's hardware- and software-related constraints were then applied to the set of candidate components identified during the search across the DoD. The final set of nine components was determined after ensuring that each met a subset of the criteria for the Demonstration and could operate in concert with the aforementioned scenario. The application architecture for the Large Scale Demonstration is shown in Figure 15.

Each of the Demonstration's components, including their respective sources and roles, are discussed below.

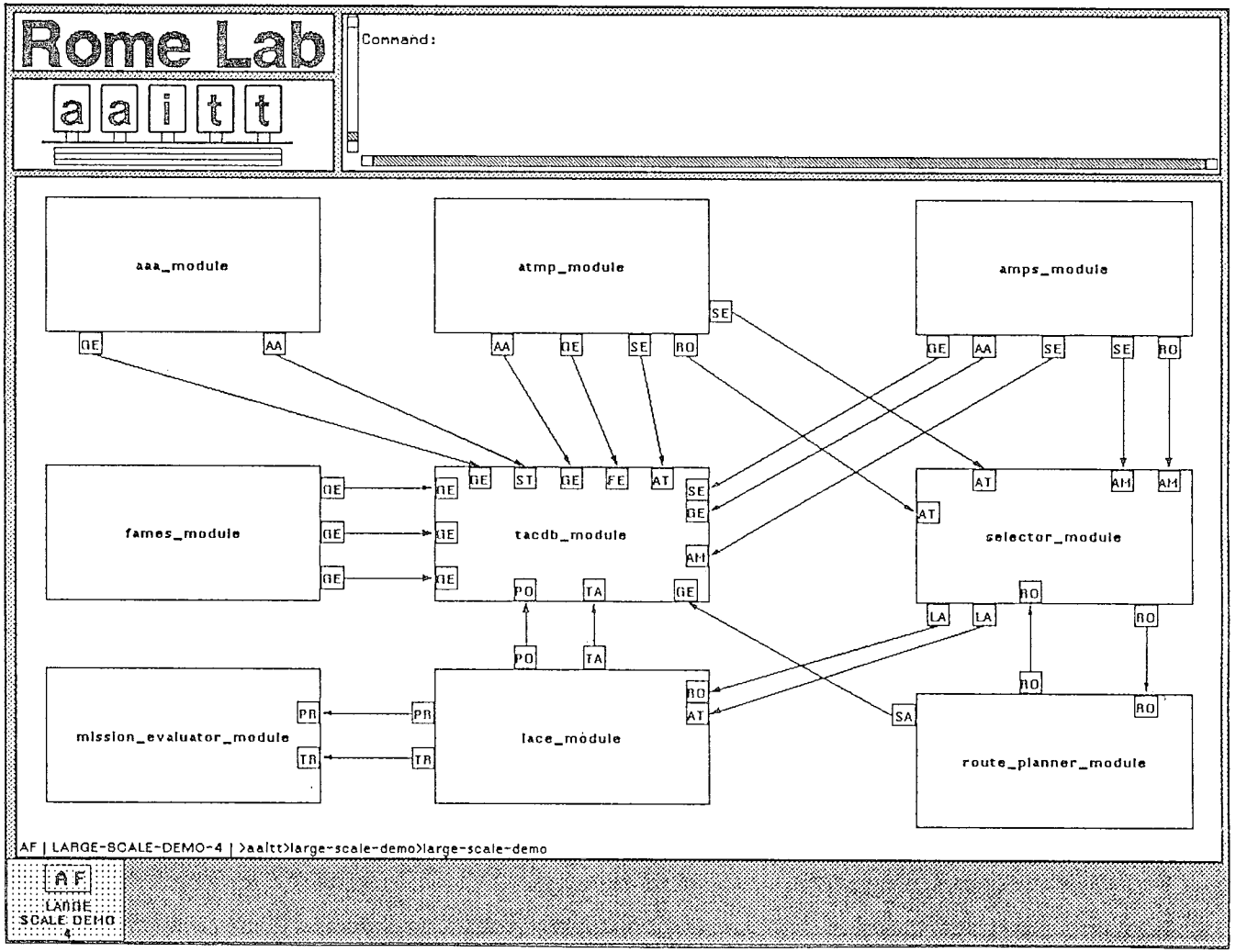


Figure 15. Large-Scale Demonstration's Application Architecture

Rome Laboratory was the source of five of the Large Scale Demonstration's nine components: TAC-DB, AMPS, ATMP, LACE, and a Route Planner. The TAC-DB database was used to maintain information regarding tactical map features, attribute updates, aircraft, beddown airbases, sortie scenarios, and SAM (Surface to Air Missile) locations. Both AMPS and ATMP (A Tactical Mission Planner) generated ATOs and optional route requests. As before, LACE was responsible for scenario execution and monitoring. The simulator also provided narrative descriptions of mission performance as well as post mission reports upon the successful completion of a mission. Finally, RL's Route Planner supplied a list of waypoints between the friendly airbase and the target based on the most currently available threat data.

Two components were acquired from the USAF's 7th Communications Group at The Pentagon. FAMES (Functional Area Manager's Expert System) used data about the number of US aircraft at the host beddown airbase to determine the type and number of personnel and transportation vehicles required to support friendly airbase operations at that site. The Airfield Attack Advisor (AAA) identified the beddown airbase, the type and number of aircraft required, recommended weapons, the number of sorties required for specified target destruction, as well as required replacement assets based upon post mission report information received.

Lockheed Martin's Advanced Technology Laboratories constructed a Selector (or Switch) Module which enabled the operator to select either the AMPS- or ATMP-generated ATO, and optionally accept the planned navigational route received from the Route Planner.

General Electric's Corporate Research and Development Center developed a sophisticated Mission Evaluator which provided planning recommendations to the operator by assessing and analyzing the currently simulated mission. The component accomplished its task using reasoning under uncertainty as well as user-developed Measures of Effectiveness.

Given today's levels of automation, most activities associated with the Large Scale Demonstration's problem-solving suite would, typically, be carried out manually by numerous personnel possessing the requisite functional expertise. The results of their efforts would be reported to and displayed manually at a Tactical Air Control Center (TACC). This process is graphically depicted in Figure 16. Successfully completing the Large Scale Demonstration showed how operations within the TACC could be automated using an integrated suite of components providing electronic coordination between the functional areas. In addition to providing users with the capability to plan an entire operation, the resulting TACC-oriented application, more importantly, also offered the ability to automatically simulate the planned missions and provide valuable feedback and recommendations to facilitate Battle Staff training. Figure 17 shows the AAITT-supported Tactical Air Control Center.

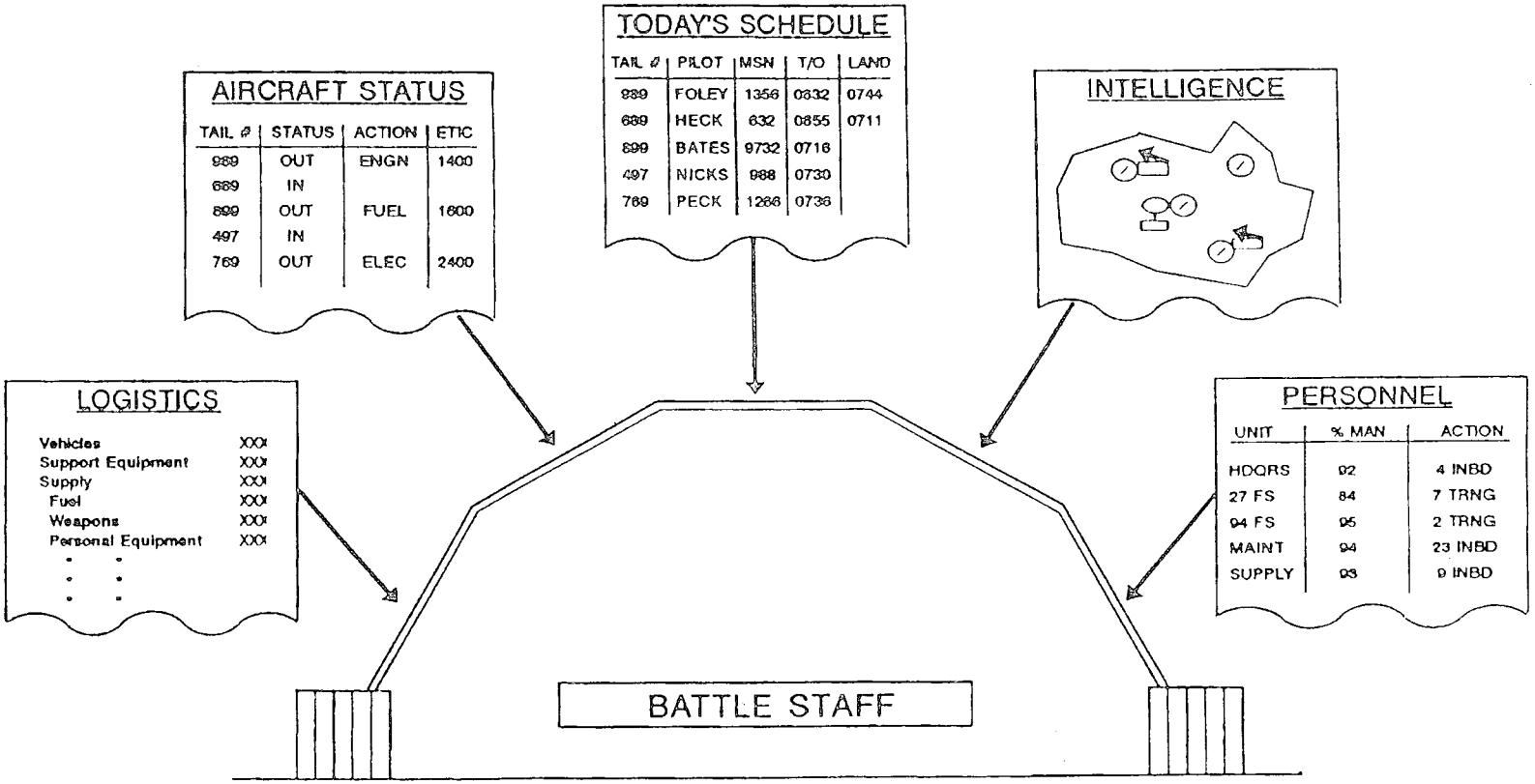


Figure 16. Manual Operations at a Tactical Air Control Center

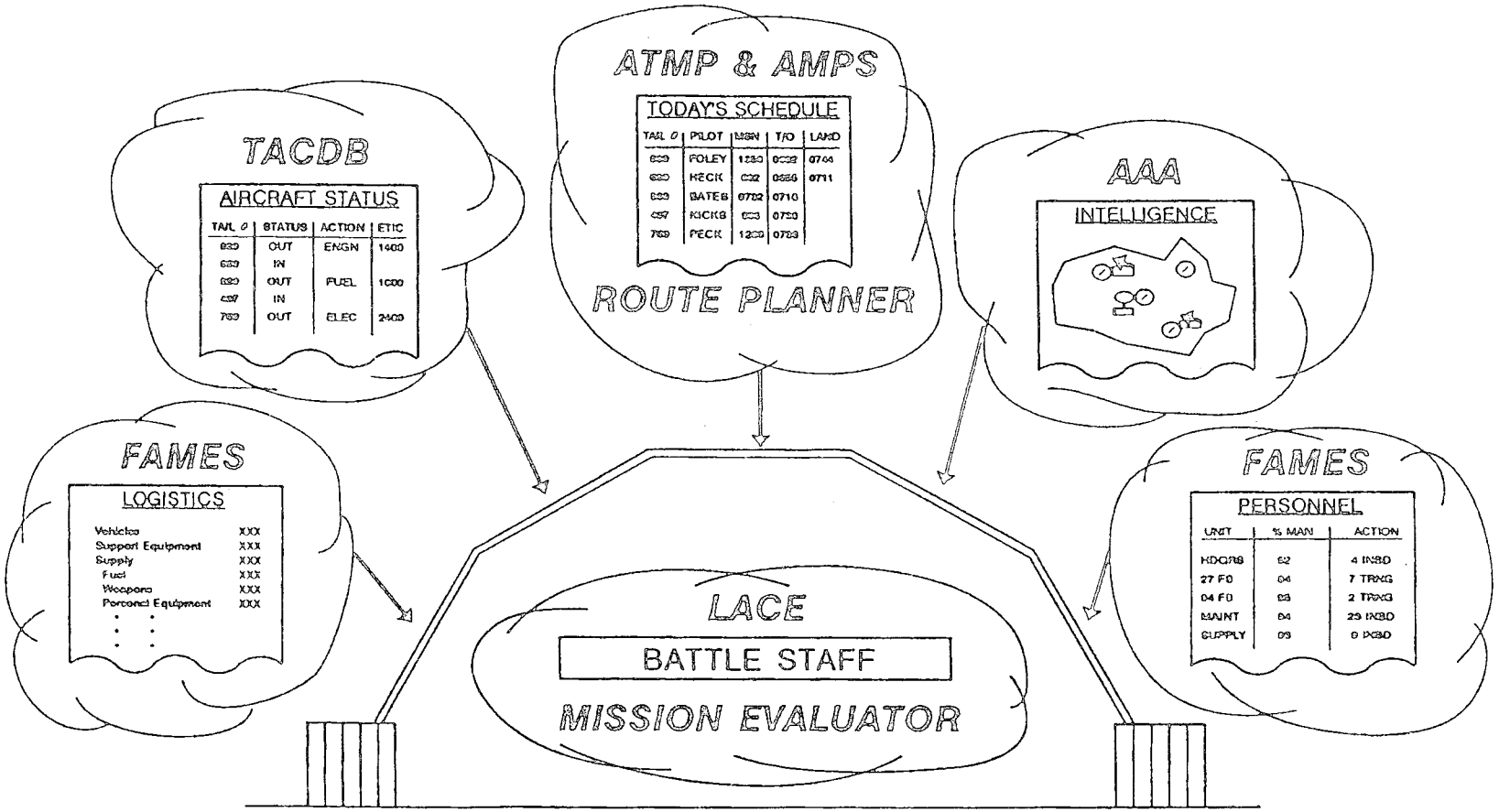


Figure 17. AAITT-Supported Tactical Air Control Center

6.3 Reusability Demonstration

Programmatic requirements for the Reusability Demonstration necessitated a demonstration illustrating that the testbed's concept and architecture were applicable across multiple domains. Thus, this demonstration's problem domain had to be relevant to the USAF, but outside of C3I. In addition, the application had to contain multiple agents and involve the use of the testbed's simulation capability. In concord with RL, a decision was made to capitalize on the USAF's investment in the fruits of the ARPA/Rome Laboratory Planning Initiative (ARPI) by using the components which comprised ARPI's second Integrated Feasibility Demonstration within the Reusability Demonstration. The three components were SOCAP, FMERG, and DART. Each of these mature packages is described next.

SOCAP (SIPE for Operations Crisis Action Planning) assists Crisis Action Planners in generating multiple Courses of Action (COAs) to the major force level. The COAs include initial plans for force phasing as well as logistics requirements. SOCAP produces a Time Phased Major Force List (TPMFL) and is based on SIPE, a knowledge-based generative planner.

FMERG (Force Module Enhancer and Requirements Generator) aids the Crisis Action Planner by retrieving and supporting the editing of pre-packaged forces from a Force Module Library, which includes combat support provisions for the specific Course of Action specified. FMERG produces a full TPFDD (Time Phased Force Deployment Data).

DART (Dynamic Analysis and Replanning Tool) is a relational database and closely-coupled simulation capability that facilitates the analysis of TPFDDs using simulation as well as modification or replanning.

In general, integrating these components allows a Crisis Action Planner to determine the transportational feasibility of a SOCAP-generated COA. Once SOCAP is given the specific mission objectives, politico/military guidance, and required resources, it develops a COA. FMERG then takes the SOCAP-generated TPMFL, which is a skeletal Deployment Plan, and creates a standard TPFDD by instantiating both specific force units as well as accompanying support and sustainment resources. The resulting full TPFDD is then "tested" by DART using transportation feasibility estimators. Shortfalls are fed back to SOCAP, which can modify the plan in an attempt to overcome any transportation problems identified in the simulation.

The completed application was established five months ahead of schedule. Additionally, due to the AAITT's graphical interface and powerful, automated tools for embedding components, the integration effort was accomplished in only five days and required no modifications to the testbed. The application and module models developed for the Demonstration are shown in Figure 18.

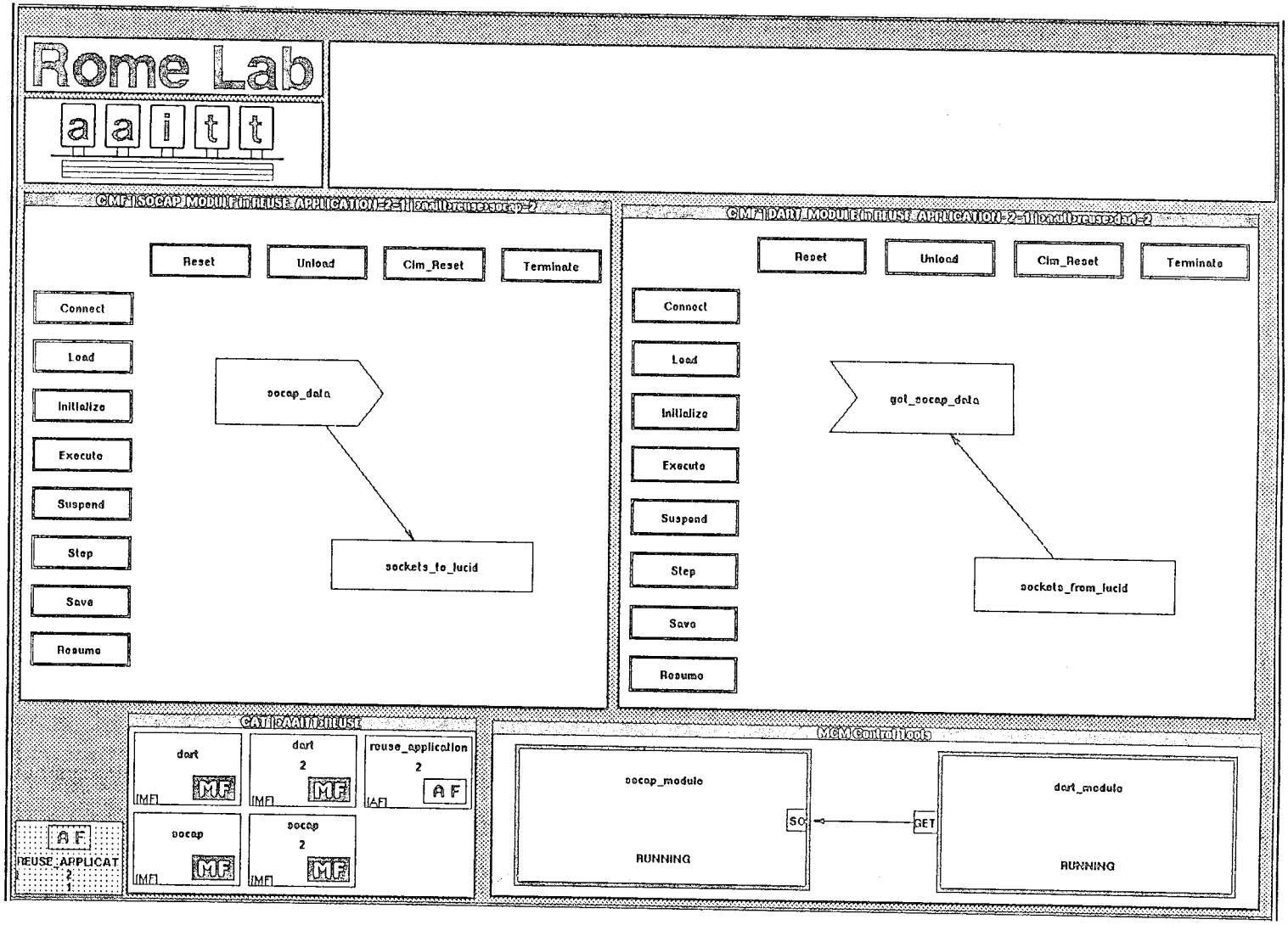
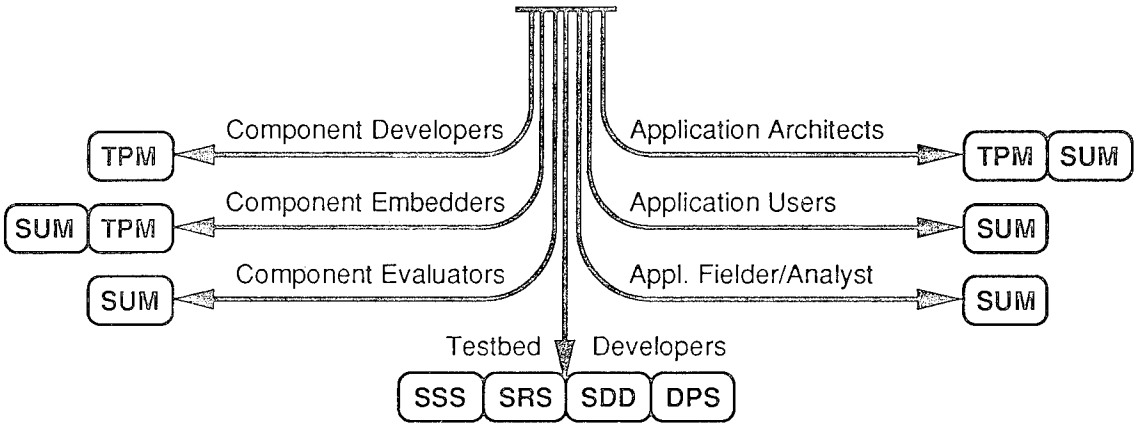


Figure 18. Reusability Demonstration's Module and Application Models

During the Reusability Demonstration, an additional, important capability was presented. The MCM Workstation had been ported from a special-purpose, Symbolics computer, and was running on a general-purpose, Sun platform. This considerably broadened the AAITT's potential user base.

7 Where To Find More Information

Tailored DoD-STD-2167A documentation was prepared to preserve various aspects of the AAITT program. The documentation serves as a rich source of information to explore the testbed in greater depth. Figure 19 provides a road map to locate the appropriate document covering a particular topic of interest.



Road Map Key

Symbol	Document	Information about
SSS	System Segment Specification	Top-Level Overview and Requirements
SRS	Software Requirements Specification	Detailed System Requirements
SDD	Software Design Document	Detailed Design Data
DPS	Distributed Processing Substrate Analysis	Distributed Communications Software Analysis
SUM	Software User's Manual	Modeling, Control, and Monitoring
TPM	Testbed Programmer's Manual	Communication and Message-Level Programming

Figure 19. AAITT Documentation Road Map

Copies of these documents can be obtained from the contracting agency or as directed by the contracting officer. Documents should be formally referred to as follows:

AAITT-A003, *Technical Information Report (DPS Analysis),* February 1991.

AAITT-A004, *Software Design Document for the Advanced Artificial Intelligence Technology Testbed CSCI,* October 1992.

AAITT-A005, *Segment Specification for the Advanced Artificial Intelligence Technology Testbed,* April 1991.

AAITT-A006, *Software Requirements Specification for the Advanced Artificial Intelligence Technology Testbed CSCI, (Revision A),* June 1992.

AAITT-A009, *Software User's Manual for the Advanced Artificial Intelligence Technology Testbed CSCI,* June 1991.

AAITT-A010, *Technical Information Report (Testbed Programmer's Manual),* June 1991.

The following document was referenced in this report. As before, copies can be obtained from the contracting agency or as directed by the contracting officer.

Cooperating Expert Systems (COPES) Final Report, submitted by Grumman Corporation to United States Air Force Rome Air Development Center under Government Contract Number F30602-88-D-0004, June 27, 1989.

8 Results and Discussion

The principal result arising from the AAITT program is a successful, high-quality, laboratory testbed which embodies a structured development paradigm and associated toolkit to support the design, analysis, integration, evaluation, and execution of large distributed software systems. As demonstrated, use of the AAITT can significantly decrease the software integration costs associated with these complex systems.

Overall, quality was increased and cost was decreased by leveraging two mature capabilities. The testbed's DPS, based on the Rome Laboratory-sponsored Cronus distributed object environment, is a flexible software backplane that accepts an arbitrary number of stand-alone components and provides heterogeneous interprocessor support for the concurrent execution of multi-agent applications. Similarly, the ARPA-sponsored ABE systems engineering tool acted as the foundation of the AAITT's Modeling, Control, and Monitoring Workstation, a graphical, module-oriented programming facility for embedding, integrating, executing, and analyzing independently-developed, cooperative, problem-solving software components.

The AAITT program's derived results can be categorized in two ways. Some outcomes directly support the needs of the DoD technical community. Others can be utilized by operational users. The advantages available to each group are discussed below. It is interesting to note that different aspects of the same result may provide benefits to both groups.

8.1 Technical Results

As more and more individuals recognize and seek to utilize the inherent power found within distributed systems, software developers will become increasingly dependent on the ability to assemble solutions from disparate pieces. It is here that the AAITT's strengths lie. The testbed's support for the complex process of distributed system integration, debugging, and evaluation, as well as the elimination of the need to extensively re-engineer components prior to integration, allows scarce time and resources to be prudently focused around component development. This focus is paramount because the completed application's problem-solving capabilities are found within its components, not within the distributed infrastructure which permits the components to interact. In the absence of the AAITT, many efforts have shortchanged component development to otherwise concentrate on integration.

The testbed's flexibility allows knowledge-based and conventional components to exist within the same application. Developers are able to selectively employ the most appropriate paradigms. Force-fitting a solution to a technology is no longer required.

The AAITT transforms graphical models into customizable code skeletons to facilitate component embedding and supports the development of various problem-solving and control strategies. In most cases, low-level details are handled automatically. Component Embedders and Application Architects are immediately productive and able to apply their energies where it is most needed. In addition, this approach has introduced a level of abstraction, making future upgrades to, or replacement of, the DPS easily accomplished. Changes will result in minimal disruption to existing component and application models.

One of the most important and interesting aspects of the testbed's capabilities comes to light once an application has been initially established. Application monitoring, capturing measurements at levels ranging from resource usage to domain-dependent solution quality, when coupled with the ability to rapidly alter the nature of component interactions, allows numerous, repeatable experiments to be performed. These experiments allow users to fine-tune their applications through the use of empirical data. Decreasing the effort and cost associated with experimentation improves quality by encouraging developers to not be satisfied with their first answer, seek better solutions, and not pursue alternatives based simply on intuition. An excellent example of this process is documented in Appendix A, Instrumented Domain Experiments.

Component reuse and the development of new solutions by adapting past successes is further facilitated by the testbed's module and application catalogues.

The technical community can also take advantage of the AAITT's approach to human-computer interaction for modeling, control, and monitoring. Initially, during the modeling phase, "the picture is the program." Next, the graphical model constructed during the application development process is the same one used to control application execution. Finally, monitoring activities are supported by a user-configurable graphic interface.

8.2 Operational Results

Unanticipated conflicts will become the norm now that we have witnessed the end of the Cold War. These hostilities are often termed "come as you are" wars because one does not have the luxury of time to methodically prepare for them. They can occur at anytime and anywhere, generally planned by the enemy to be the most inconvenient to the US. The ability to rapidly assemble and (re)configure software applications to automate and facilitate the decision making process is an essential part of responding effectively to new or escalating situations.

Successfully establishing each of the AAITT program's applications clearly demonstrated that the testbed can provide this capability. In every case, interoperable sets of software components were established quickly, although

the application's elements were never initially designed or developed to function together. Completed applications were not limited to a single USAF domain and integrated problem-solving aids supporting various functional areas.

Applications developed using the AAITT do not represent custom, inflexible "point solutions." Constituent components can be introduced, removed, or combined as desired, offering the opportunity to constantly adapt, improve, and scale-up the functionality provided to the Warfighter with minimal disruption to existing capabilities. The ability to intermix conventional and knowledge-based modules within a single application allows today's legacy systems to take advantage of tomorrow's advanced technology.

In addition to the obvious time and cost benefits that result from eliminating the need to extensively re-engineer components during the embedding and integration process when using the AAITT, quality and productivity gains will accrue to the application's operational users. In many cases, the re-engineering process can introduce both anomalies and sources of failure into a component. Retesting is expensive and, more importantly, not guaranteed to uncover new problems. The heat of battle is not the time for surprises to come to light. It is also crucial for users, particularly those operating in critical situations, to continue utilizing their "native" decision aids. Otherwise, productivity will suffer and the opportunity to commit errors of both omission and commission will increase. Encapsulation, as supported by the testbed, permits components to retain their original "look and feel."

The development model embodied by the AAITT is also consistent with the DoD's ongoing acquisition initiatives. The testbed supports the recent emphasis placed on the incorporation and reuse of both commercial- and government-off-the-shelf (COTS & GOTS) components to construct systems. Furthermore, the AAITT's entire iterative modeling, control, and monitoring paradigm is centered around the notion of realizing solutions using a rapid prototyping strategy, where results are placed into the hands of users as soon as possible for evaluation and feedback. Finally, the testbed facilitates evaluations which can be used to control the process of judiciously introducing only sufficiently-mature technology into operational systems.

9 Conclusions

The most important conclusion which can be reached from this effort is that the AAITT program's numerous successes conclusively validated the software integration paradigm embodied within the testbed. An innovative approach to encapsulation, using graphically-specified and automatically-generated wrappers, was repeatedly demonstrated to be a cost-effective means of building distributed software applications without the need for extensive component re-engineering. Accomplishing the same using more expensive and complex strategies, such as reimplementation or universal data/information interpreters, is no longer the only option available to developers.

The testbed's paradigm and toolkit resulted in dramatic productivity gains and equivalent decreases in integration costs for tasks of the same magnitude. It is interesting to note that during the development of the applications for the Large Scale and Reusability Demonstrations, considerably more effort was expended identifying and acquiring suitably mature decision aids than was spent actually embedding and integrating the selected components.

A wide range of accomplishments laid the groundwork for the successes which validated the AAITT's software integration paradigm. These accomplishments are grouped by functional area and enumerated below.

9.1 Distributed Processing Substrate

☆ Cronus, an existing commercial distributed object environment, was provided with a mouse-and-menu interface to improve its usability and operability. Without the interface, Cronus is simply a procedural, message-driven system. In the future, this layer of the AAITT's DPS may be replaced with a CORBA (Common Object Request Broker Architecture)-compliant product, as availability allows. The DPS was intentionally implemented using a level of abstraction to facilitate this type of interchange. Among other things, a CORBA-compliant system would offer the added feature of maintaining persistent objects within the testbed.

9.2 Modeling

☆ A graphical notation for the specification of a component's communication interface to other components within a distributed heterogeneous system was developed for the testbed. This notation was then mapped into an equivalent, tangible processing implementation. When combined, the notation and mapping form the foundation of the visual modeling environment used to construct module wrappers.

☆ The communication models developed using the aforementioned graphical notation are subsequently used to ensure module compatibility during application modeling.

☆ Modeling within the testbed does not depend on the actual existence of the components at the time they are being modeled, i.e., the use of stubs is supported.

☆ At the outset of the program, a goal was established to provide tools supporting a Component Interface Manager's capability to communicate with its associated component. Although a library of routines implementing a (UNIX) socket-based communication scheme was developed for this need, this area deserves additional attention.

☆ The AAITT's catalogue capabilities facilitate the organization, management, and reuse of module and application models, as well as related metrics.

☆ During the course of developing the AAITT, assembling the applications used to demonstrate the testbed, conducting the AAITT Training Course, and, in particular, performing the testbed-supported analyses that acted as the foundation of the Instrumented Domain Experiment discussed in Appendix A, the power placed in the hands of users by the testbed's high level of flexibility became increasingly evident. Testbed Developers, Component Embedders, and Application Architects alike, were able to transparently redistribute modules, adapt/reuse existing components and applications, as well as rapidly alter control strategies using the testbed's graphical Frameworks. These capabilities cannot be overemphasized.

9.3 Code Generation

☆ The AAITT's code generators significantly advanced the state-of-the-art in automatic code generation, based on their ability to transform complex, graphical models into executable images, distributable across a network of heterogeneous machines.

☆ Compiling a distributed heterogeneous system is an intricate and time-consuming task. The AAITT incorporates both an automated mechanism to manage this process for the user as well as an incremental compilation capability to conserve resources and increase productivity by recompiling only the minimum required to reflect the changes which have been made.

9.4 Control

☆ A state diagram for distributed heterogeneous systems, accommodating both conventional and knowledge-based components, was developed and realized within the testbed to provide users with the means to achieve any desired level of control in a flexible, yet disciplined, fashion.

☆ The AAITT's control software provides visual feedback to the user so that an application's current state(s) can be easily discerned.

☆ The testbed's module-to-host assignment process emerged as a mouse-and-menu task during this program. By leveraging Cronus' capabilities, assignments can be transparently handled within the same processor family.

9.5 Monitoring

☆ Under the auspices of the AAITT program, a methodology was established for synchronizing the time stamps of logs captured on distributed processors, collecting those logs, and merging their contents to support module and application analysis.

☆ Collected logs are intentionally maintained in a simple, ASCII representation so that the data can be easily filtered and introduced into other data analysis tools such as spreadsheets and statistical packages. This process was used to establish a loose coupling between the AAITT and the University of Massachusetts' CLASP (CommonLISP Analytical Statistics Package) software and facilitate the analyses conducted for the Instrumented Domain Experiment discussed in Appendix A.

☆ Each testbed-resident module is provided with built-in measurements capable of capturing low-level resource utilization metrics, such as CPU-usage and memory swapping data.

☆ Information regarding intra-module communication can also be captured using the testbed. Message content and timing can be reviewed using a graphical, "logic analyzer"-style display that aids debugging.

☆ The issue of domain-specific metrics was also addressed during the AAITT program. Although a sophisticated "Evaluator Module" was developed for the Large-Scale Demonstration to examine the topic of solution-quality, this area could benefit from additional attention and research.

9.6 Debugging

☆ In general, building a distributed system is very complex and chaotic. The development process begins as a two-dimensional situation and frequently escalates into a three-dimensional problem. Successfully debugging these systems is a skill which often approaches an art. The team's extensive experience constructing these types of systems led to the incorporation of many control and monitoring features specifically designed to aid with debugging, including breakpoints, logging taps, CIM-query capabilities, long-form logs, a dynamic message facility, as well as a state transition strategy permitting modules to be paused and single-stepped.

The resulting AAITT raises distributed system development to the next level.

10 Recommendations

The AAITT is a high-quality laboratory testbed, capable of filling a critical technology role for both technical and operational users. It provides a vital software integration and evaluation toolkit of immediate utility to the DoD, in general, and the USAF, in particular. It is clear that the requirement for and the value of these capabilities is ongoing and will continue to increase in the future. Why? The number of independently-developed, conventional and knowledge-based automation aids, offering problem-solving support across a wide array of functional areas, will explode. The benefits of stand-alone components are limited, at best. True, integrated decision support will occur when these pieces can be rapidly and cost-effectively combined into interoperable, situation-specific applications which meet a commander's operational or scientist's technical needs. A commercial equivalent does not exist at this time. Thus, the Lockheed Martin team recommends that an AAITT productization effort be undertaken to move the testbed out of the laboratory and in to widespread use.

A parallel, dual-track strategy is suggested, where extensive testbed evaluation would be conducted concurrently with an iterative development effort.

Assessing the AAITT is best accomplished by placing successive versions of it into the hands of a wide range of technical and operational users, training them in its use, providing support, and periodically following-up to solicit feedback on its strengths and weaknesses. This invaluable feedback would then be incorporated into the requirement set for the next development cycle. The set of users would potentially include participants in field exercises and interoperability demonstrations; organizations interested in establishing "software test ranges," permitting the insertion and assessment of new components within a standard evaluation suite; efforts to construct "anchor desks" incorporating capabilities from existing, legacy infrastructures; as well as initiatives strongly focused around component development, where frequent integration and ongoing experimentation may be desired but unaffordable in the absence of a powerful tool such as the AAITT.

In addition to prudently responding to the suggestions obtained from user feedback, the development effort must begin by defining the form of the final, productized testbed. A number of the issues which must be addressed are briefly discussed below. In general, there will need to be an increase in emphasis on meeting the specific requirements of the operational community.

Users..... The testbed is presently oriented to supporting the technical user community. However, in contrast to Application Users, who depend on the Component Developers behind each constituent decision aid, the fielded AAITT product must support the needs of the SC/J6 staff assigned to assemble mission-specific decision support applications.

Hardware..... The range of hardware capable of hosting either the AAITT or individual components within testbed-resident applications must be expanded. Particular attention should be paid to the DoD's installed base of personal computers, including laptops.

Software..... Support for components implemented in the C or LISP languages currently exists explicitly within the AAITT. Other languages are implicitly supported; the only requirement being the ability to link the "foreign" language with any of the supported languages. Additionally, virtually all candidate components can be treated as "black boxes" in cases where a less dependent level of coupling is acceptable or the Component Embedder simply has no other choice. Thus, although few restrictions exist, augmenting the AAITT with explicit support for additional languages such as Ada as well as making the Distributed Processing Substrate CORBA-compliant may be both desirable and worth the cost.

Communications..... Communication is a crucial capability for operational users in particular. The testbed must be compatible with and leverage the advances envisioned for the Global Grid and National Information Infrastructure, as well as the means to access various tactical links using the appropriate protocols and formats.

Security..... Protecting information at multiple classification levels as it is passed between components has not been addressed to-date within the AAITT program. This crucial capability must become an integral portion of the MCM Workstation, the Distributed Processing Substrate, as well as each module's wrapper.

Portability..... This issue spans, and is dependent on, a number of the topics presented above. However, it warrants special mention because operational users distributed world-wide must be able to engage in problem-solving activities regardless of location, continue operating while en route, and maintain seamless contact despite transportational transitions.

User-Friendliness..... Although the AAITT already incorporates a highly-graphical, context-driven human-computer interface within its frameworks and makes extensive use of

point-and-click menus to guide user selections, the testbed needs to be exercised by a wider audience to identify areas where the interface and concept of operation need improvement.

Recoverability..... The AAITT must be able to rapidly and accurately recover from unforeseen events, including computer system failures caused by battle damage as well as operator-originated errors. Adaptive fault resistant techniques currently under development hold the promise of helping in this regard.

Training..... Obtaining the full benefit of the testbed's capabilities would be better ensured by offering users on-line tutorial and help information, as well as knowledge-based assistance for debugging and application performance analysis.

An item orthogonal to these issues is the need to provide distributed modeling, control, and monitoring capabilities within the testbed. This would allow multiple, non-contending instantiations of the MCM Workstation to execute simultaneously across a wide-area network (WAN). Although the Distributed Processing Substrate presently supports modules communicating across a WAN, stand-alone components, Component Embedders, as well as Application Architects must all be co-located before component embedding and integration can take place. The cost to augment the AAITT with the ability to permit these procedures to occur remotely would be easily dwarfed by the savings accrued from eliminating required travel alone.

These general requirements for a productized testbed need to be confirmed with a full, representative cross section of users, who will undoubtedly provide valuable amplifying details. Finally, once the AAITT is functionally complete, the product must be "bulletproofed," to minimize the possibility that user activity can unintentionally disrupt the testbed's operations.

This productization effort deserves immediate consideration. Leveraging the current endeavor's accomplishments and momentum into a follow-on program is vitally important to cost-effectively facilitating the insertion of decision support assistance into an operational community which continues to grow increasingly dependent on automation aids as budgets continue to shrink and commanders are forced to deal with reduced numbers of functional area experts and staff personnel. Otherwise, this technology's benefits will never reach tomorrow's Command Centers.

Appendix A Instrumented Domain Experiments

This appendix describes the results of an effort which explored the concept of "Instrumented Domain Experiments," or IDEs, successfully performed under the auspices of an engineering change to the base AAITT contract. This effort can be viewed as a two-phase undertaking — an initial investigation into the principles behind IDEs, followed by a practical application of the concept using the AAITT.

Thus, this section begins by discussing the motivation for Instrumented Domain Experiments. The formulation of an IDE questionnaire, distributed within government, academia, and industry, is then presented. Survey results are subsequently summarized. These responses helped the team arrive at the IDE definition offered here.

The testbed was used to perform an IDE. The conduct and results of the IDE are described next, and include a discussion of the loose-coupling which was achieved between the AAITT and the University of Massachusetts' CLASP package. Finally, the appendix concludes by presenting the specific support which the testbed provides for IDEs, affirming our belief that the AAITT can act as an effective foundation for the experimentation activity which forms the heart of the Instrumented Domain Experiment process.

A.1 Background

One of the obstacles to developing software applications for complex, real-world problems has been the difficulty of evaluating candidate technologies as well as prototype systems to determine their promise or degree of success. Determining the answer to questions such as the degree of a technology's scalability or whether one technology holds more promise than another for a given application has proven to be a very difficult proposition. Frequently, software technologies end up being evaluated on the basis of prototype demonstrations, often using a single, possibly unrepresentative, input scenario. Thus, the evaluator is hard-pressed to judge the depth and robustness of the prototype's functionality. Since the current limitations of the prototype are usually avoided in these demonstrations, it is also difficult to assess progress and judge the potential for further development.

There is an increasing need to both better focus application-oriented software R&D (Research and Development) and achieve faster, more cost-effective technology insertion. Thus, there is a corresponding increase in the importance of improving the ability to evaluate technologies for use in real-world domain applications. The goal is to define an evaluation methodology which will support the more systematic and thorough evaluation of new technologies, specifically, the potential utility of these technologies for solving problems within application domains.

The possibility of using some type of software experiments to address these goals was raised, and the term "Instrumented Domain Experiment" was coined by Dr. Stephen Cross at ARPA to denote the concept of using software experiments to evaluate technologies for domain application. However, the issues involved and the question of exactly how one might go about conducting such experiments required additional investigation.

The term IDE indicates a focus on two key elements. First, the use of actual problem domains rather than artificial, simplistic problem spaces. Second, the employment of experimentation instead of restricted "feasibility" demonstrations.

One obstacle to reliable technology evaluation has been the common practice of demonstrating systems using inputs based on overly-simplified problems. Often, only a few examples are applied; in extreme cases, only a single instance is used. It is not unreasonable to expect that better technology evaluation would occur if systems were exercised against problems representative of the target domain's scope and complexity. However, this notion then raises a number of questions, such as:

- *"How can one determine whether or not a given problem case is representative?"*
- *"Is it necessary to use only "real" problem cases, with all of their attendant ambiguities and complications, or should problems be limited to some degree?"*
- *"If so, how does one recognize when a problem case is too limited?"*
- *"Where does one obtain acceptable problem examples?"*

Thus, although the general notion of seeking more rigorous evaluation through use of more demanding test cases seems sensible, instituting such a practice is likely to require careful thought and investigation.

Similarly, it has long been a common practice in software R&D for developers to "prove" the value of their systems by demonstrating that they, indeed, run and produce apparently correct output when exercised on the types of test cases described above. Such demonstrations often provide little in the way of useful information to the evaluator. In fact, they provide no answers to the evaluator's questions about issues such as the demonstrated system's efficiency, the quality of its output, its merits relative to other technical approaches, and so on. These kinds of questions could be addressed if one could replace the traditional feasibility demonstration with one or more rigorous experiments.

Similar to the idea of using more representative problem spaces, substituting demonstration with experimentation raises a number of practical questions, such as:

- *"What kinds of questions should be experimentally tested?"*
- *"How can these questions be formulated as testable hypotheses?"*
- *"What measurements should be used to evaluate a hypothesis?"*
- *"Must all 'experiments' follow the classical model by providing an explicitly formulated hypothesis to test?"*
- *"In summary, what are the effects of focusing the general concept of software experimentation on the evaluation of software technologies for domain applications?"*

Ultimately, the answers to all of these questions must be answered through experience. However, the team concluded that initial efforts to gain such experience would be more fruitful if preceded by an analysis of the IDE concept and its attendant issues. Thus, some of the issues which must be addressed in applying experimentation within this context are presented next, and the discussion then goes on to describe candidate definitions of an IDE and an associated methodology.

A.2 Approach

Beginning with the general idea of using software experimentation to evaluate technology for domain applications, several areas for analysis were identified:

- The overall goals for IDEs
- A candidate IDE methodology
- The relationship between IDEs and the software engineering process

First, it is interesting to consider the underlying reasons for performing IDEs. Software experiments performed for basic research would, presumably, have the goal of establishing the fundamental principles and properties of, for instance, planning or computer learning. On the other hand, the types of goals that would make sense for IDEs would likely be different. A variety of interested parties are likely to be involved in an IDE, including operational Application Users, Application Architects, (Government) sponsors, as well as the individuals conducting the IDE. Each of these parties will, in all likelihood, be pursuing different goals. Thus, identifying likely categories of IDE goals was considered the first important area of investigation.

The question of how an IDE should be conducted is also critical. Issues which must be addressed here include the nature of the input data to be used, approaches to measuring results, and the characteristics of appropriate

experimental hypotheses. Therefore, formulating an IDE methodology was identified as the second area of investigation.

Finally, it is important to consider when, in general, IDEs should be performed. In particular, the team was interested in identifying the best insertion point for IDEs within the software development process. This question required attention because the answer might depend, at least in part, on the particular software development model, e.g. "waterfall" versus "spiral," being adopted.

In summary, the intention of the preliminary analysis was to establish a starting position on the three basic questions of why, how, and when to perform IDEs.

The team created a questionnaire addressing issues in each of the three topic areas to facilitate data gathering. The questionnaire was distributed within the operational and technical communities for advanced software R&D. Responses were compared and analyzed to develop recommendations for IDE goals, an IDE methodology, and the place for IDEs in the software development cycle. Survey responses will be summarized below, along with a discussion of the team's analysis and recommendations.

A.3 Initial Methodology

A strawman IDE methodology was defined and inserted into the questionnaire to stimulate the canvassing process. It consisted of the following steps:

1. Produce the specification of a particular experiment, including

    a. The application suite selected for evaluation;
    b. The overall goal(s) of the experiment;
    c. The evaluation metrics to be applied;
    d. The rationale for the choice of metrics;
    e. The measurement strategy for the metrics;
    f. The data-capture strategy for the measurements;
    g. The inputs to drive system execution; and
    h. The rationale for choice of input set.

2. Introduce the application suite into the instrumentation facility by embedding and integrating the application's constituent components.

3. Implement or integrate the experiment's required data-capture mechanisms.

4. Conduct the experiment under controlled conditions.

5. Analyze the data captured during the conduct of the experiment.

6. Present analytical results to the targeted audience for review.

A.4 Questionnaire Results

The completed questionnaire was distributed to individuals in government, academia, and industry. Table A-1, below, characterizes those who responded.

Category	Responding Organization(s)
Academic Research	◆ University of Massachusetts
Industrial Research (Both defense and non-defense)	◆ Lockheed Martin Advanced Technology Laboratories ◆ GE Corporate Research & Development ◆ Teknowledge Federal Systems
DoD Developers	◆ Headquarters, US Army AI Center
DoD Sponsors and Operational Users	◆ USAF Rome Laboratory ◆ US Army Intelligence Center ◆ Joint National Intelligence Development Staff
DoD Verification and Validation (V&V) / Testing	◆ USAF 7th Communications Group ◆ Air Combat Command / 1912 Computer Systems Group ◆ Ogden Air Logistics Center / SCTE (Computer Support Group)

Table A-1. IDE Questionnaire Respondents

The questionnaire contained approximately 20 questions distributed among the topics of IDE goals, methodology, and relationship to the software development process. Below, the questions which produced the most substantive responses are presented, along with specific examples or characterizations of the answers received.

Q: *What types of questions would you want answered as the outcome of an evaluation?*

A: Sponsors and operational users expressed an interest in determining whether operational needs were being satisfied as well as evaluating added value. As might be expected, the test and evaluation community wanted to know whether the system met user requirements. Multiple themes from "Did it work?" to "What are the data requirements?" were raised by industry developers. DoD developers also felt that various issues should be addressed, but felt that the specifics would be a function of the iterative development process. The sense from the academic community was that it was important to consider whether the science or technology was being advanced, using both sensitivity and parametric estimation studies.

Q: *What issues should not be addressed by these experiments?*

A: The consensus was that any issue should be fair game. However, several interesting exceptions arose. An individual from the sponsor and operations group felt that it was crucial to agree on a subjective evaluation scale before considering qualitative questions. One industry developer believed that the presence or absence of a specific technology should be off-limits, while an academic researcher wanted to avoid treating the experiments as "bake-offs" requiring clear-cut winners and losers.

Q: *What kinds of metrics would assist in meeting your evaluation goals?*

A: Due to the clear relationship between experiment goals and metrics, responses here paralleled the answers received to the goals portion of the questionnaire. Sponsors and operational users wanted metrics that measure value-added for the user. Respondents in the evaluation and testing groups uniformly focused on users as the best source of metrics.

Q: *Would the experiment be scenario driven? Would more than one be used?*

A: Although nearly everyone favored the use of scenarios, the term was not universally interpreted in the same way. While some believed that a scenario was an artificially-created entity, others felt that scenarios were datasets captured from actual operations or exercises. However, there was general agreement that it was largely infeasible to prepare multiple, real-data scenarios.

Q: *How would one demonstrate the typicality or generality of the scenario(s) or other inputs?*

A: The answers to this question gave rise to the notion of a certifying authority, an individual or organization associated with the application domain, capable of authenticating the scenario or other inputs. The result would be scenarios that are user-defined; identified as typical by a domain expert; possibly of historical significance; and/or DoD- or Service-validated. It would be the responsibility of the certifying authority to characterize the overall bounds of the scenario space as well as the coverage of the space afforded by any particular scenario.

Q: *Should the experiment be done within a real world or "near-real" world context, e.g., as part of a planned military exercise or installed on a factory floor?*

A: Most respondents believed that there would be a direct correlation between the authenticity of an experiment's context and its value. It is interesting to note that members of the sponsor and operational user groups, although agreeing that it was preferable to conduct these experiments within a real world context, also felt that some experiments should be carried out initially in a near-real world situation and subsequently transitioned into the real world. A glimpse into the reasoning behind the latter, phased approach can be found next.

Q: *Have you participated in evaluations of this kind? If so, what was good and what was bad about them?*

A: One answer deserves particular consideration. The respondent, possessing experience with these types of evaluations during military exercises, liked the intensity of the user-developer interaction during the evaluations but cautioned that prototypes frequently fail because of known limitations or unexpected changes in exercise conditions. However, although the reasons behind these failures are frequently well understood and known beforehand, one does not get a second chance to make a good first impression. Thus, a possible outcome of these failures is irreparable damage to the user's confidence in the capability being evaluated.

Q: *Where would an IDE fit in the software development process?*

A: The consensus was that, regardless of the particular development model being employed, experimentation could be valuably applied throughout the entire process.

Q: *Would the IDE replace or add to the steps in the software engineering cycle? Where would the IDE be added, or if steps would be replaced, which ones?*

A: There was no clear agreement among the respondents to this question. Some individuals felt that IDEs might replace portions of the testing process. Others believed that IDEs address issues other than testing, such as the definition of requirements.

Q: *What fraction of project resources should be applied to IDEs and from where should it be taken?*

A: Answers here covered the entire spectrum, from "insignificant" to "half of the effort."

The implications of these responses are examined next.

A.5 Analysis

A number of tentative conclusions regarding IDE goals, the IDE methodology, and the relationship between IDEs and the software engineering process, can be drawn from the answers provided by our respondents.

A.5.1 IDE Goals

First, with regard to IDE goals, the team concluded that the IDE must be focused on evaluating the technology's ability to satisfy operational needs. This means that IDEs should not be seen as vehicles for the advancement of basic research. Emphasizing operational needs as an evaluation driver is a natural consequence of the IDE's stated purpose as a tool for evaluating technology in application domains. However, it is not uncommon for researchers to feel that one should be able to both advance the science and satisfy application needs at the same time. The academic respondent's answers and associated comments express this belief. In contrast, all of the other respondents emphasized the importance of determining whether the technology met

operational needs. They used such terms as "requirements," "user-determined questions," and "operational needs." Thus, evaluation against operational needs as opposed to evaluation of technology advancement represent generally divergent goals.

To illustrate this divergence, consider a case where two planning technologies, referred to here as A and B, are to be evaluated. Suppose these technologies are evaluated by performing one or more experiments in a certain application domain. Furthermore, suppose it is determined that technology A is significantly faster than technology B. Such a result may be of vital interest to the researcher, but would not be important in evaluating the technology within the chosen domain unless B failed to meet a time requirement imposed by an operational need within that domain. In other words, if the goal is to improve the capability of those who plan in the chosen domain, A's speed advantage would not be a sufficient reason to spend resources on developing A rather than B. On the other hand, if the goal is to determine basic differences between A and B, the speed difference is important, and one might well be justified in expending additional resources to, for example, attempt to identify the reason(s) for the difference.

Thus, the basic purpose of IDEs is to reduce the set of possible experiment goals down to those which will aid in ascertaining the relationship between the technologies to be evaluated and the application domain's operational needs.

A.5.2 IDE Methodology

Respondents to the questionnaire also provided useful insights to methodology issues. Several points regarding the specification of metrics, evaluation criteria, scenarios, and experimental hypotheses were made.

It was stated that it is crucial to agree on a subjective evaluation scale before considering qualitative questions. The academic respondent stated that little can be said, in general, about issues such as metrics because the critical details depend on specific circumstances. However, respondents from the Sponsor and Operational as well as the Testing and Evaluation communities emphasized the importance of both measuring value-added from the user's point of view and looking to the user as the definitive source of metrics. Several important conclusions may be drawn from these points.

First, the user and his/her operational needs must be the source of IDE evaluation criteria and metrics. As discussed earlier, factors such as a software system's execution speed are only significant in the context of an IDE to the extent that the user's speed requirements are significant in the given domain.

Second, evaluation criteria and their corresponding metrics, whether qualitative or quantitative, must be agreed to in advance by all relevant parties. This is a practical necessity because, in many cases, there may be no obvious best

choices for criteria or measurement methods. The difficulties associated with choosing a subjective scale for qualitative conditions represents an excellent example here.

Finally, if we are to take the word "Experiment" in Instrumented Domain Experiment seriously, it is important to transform the evaluation criteria drawn from operational needs into clear-cut test conditions or hypotheses. This point, although straightforward, is frequently ignored in the "feasibility demonstrations" often offered to software technology evaluators. As an example, consider the domain of logistics planning. It is insufficient to use a statement such as "There is a need to increase the speed of logistics planning..." as an evaluation criterion. By this criterion, any software system which reduces planning time is a success. Using such a standard gives rise to problems because the technology evaluator, shown that System X reduces planning time, has no basis for judging whether the reduction achieved will result in a legitimate operational benefit and/or whether that benefit would be worth System X's likely fielding costs. Clearly achieving the intention of the IDE concept in this example would require establishing concrete criteria based on precisely-defined operational needs. Thus, suppose that a study of the logistics planning process revealed that planners could revise plans quickly enough to handle most unforeseen events or changes to planning assumptions without incurring significant operational delays if the time to generate a plan could be reduced by a factor of ten. This provides a clear basis for defining an IDE test hypothesis.

Questions regarding the nature and context of IDEs input sets also yielded beneficial responses. First, respondents were asked whether experiments should be driven by a scenario, defined as a dataset representing a coherent, essentially complete instance of the type of operational activity that occurs in the chosen domain. The alternative would be to choose an artificial dataset representing only certain aspects of the domain's properties. The respondents' uniform preference for scenario-based experiments appears to be a function of their emphasis on basing evaluations around operational needs. However, the use of an artificial dataset might be appropriate for an experiment aimed at pure research. Still, it is clearly critical that the data driving the experiment must accurately represent the domain in scope and complexity when the goal is to evaluate technology for its utility in that domain.

Of course, even a scenario-based experiment will not yield reliable results unless the scenario represents a typical case from the target domain. In defining the strawman IDE methodology, the team felt that one of the key challenges would be identifying representative scenarios and demonstrating their typicality. The respondents offered the very practical solution of relying on domain expertise to overcome this hurdle.

Members of the Sponsor and Operational as well as the Test and Evaluation groups recommended that scenarios constructed from live exercises or operations, and judged to be typical by domain experts, should be employed

within IDEs. At least one respondent noted that many such scenarios already exist and are maintained throughout DoD for various applications. Although several Developer respondents raised the possibility of creating artificial scenarios, the respondents with the greatest experience in delivering technology to operational users, namely the Sponsor and Operational as well as Test and Evaluation groups, felt that the use of real scenarios is the only approach to constructing datasets which accurately represent the domain's true scope and complexity. Thus, IDEs must be driven by scenarios that are composed of real domain data and judged representative by suitable experts.

IDE context is an important factor not explicitly mentioned in the questionnaire's strawman methodology. One might reasonably assume that, since the purpose of the IDE is to evaluate technologies for real-world application, IDEs should be conducted in the real world. Although most of our respondents felt "the more real the better," the experiences of some in the Sponsor and Operational community led them to recommend the more cautious strategy of evaluating in "near-real world" conditions before venturing into the real world.

The motivation for this recommendation can be found in the comments provided by one member of this group in response to the question asking each individual to indicate whether they had ever participated in an evaluation similar to an IDE. The respondent had been involved in evaluations performed as part of live military exercises and felt that, although assessments performed under these conditions could be beneficial, they also presented serious risks because the unpredictability of live operations, even carefully planned exercises, frequently resulted in circumstances which caused the prototype under evaluation to fail in an unfavorable way. For example, unexpected conditions might induce a failure due to previously known limitations of the prototype or necessitate that the prototype be used in an unintended or unconventional manner. Each successive failure can further decrease the willingness of operational users to accept a system. Examining this observation from the perspective of trying to replace demonstration with experimentation, it is clear that the underlying problem is lack of control. That is, one important characteristic differentiating experimentation from other forms of assessment is the ability to control the conditions under which the evaluation is performed. If conditions are controlled, any failure of the prototype represents additional data from which conclusions can be drawn. Otherwise, failures may be due to either previously known deficiencies or indeterminable reasons. In either case, nothing additional has been learned about the technology under evaluation. Therefore, although IDEs should be performed using real world data as discussed earlier, they should only be performed under laboratory conditions and not conducted in either the real world or the near-real world of exercises.

A.5.3 Relationship between IDEs and Software Development Models

The final major area of investigation is the relationship between IDEs and software development models. Most of the respondents used some form of iterative development model as a frame of reference, but some worked from the waterfall-style model. Surprisingly, regardless of the assumed development model, virtually all of the respondents felt that experimentation would be employed throughout the development process. One could certainly envision IDEs performed as part of each prototype cycle under an iterative development model. However, respondents who assumed the use of a waterfall model also believed that IDEs could be conducted early in the development process to assist with phases such as requirements definition.

It is important to note that performing an IDE in the early stages of waterfall-oriented development requires either a simulator or an adaptation to the development model because the IDE depends on the existence of some software to act as the subject of the experiment. The simulator is used to approximate the behavior of the intended final capability during the IDE. The adaptation alternative is similar since, in a digression from the model, an early prototype of the desired system is constructed. Either approach involves additional cost and blurs the distinction between the two models. Thus, the team concluded that IDEs are best used with iterative development models.

There was no consensus about whether IDEs would supplement or partially replace other forms of test and evaluation. This disagreement may have been the source of the wide range of answers provided to the question asking about the relative magnitude of resources which should be applied against IDEs. Despite this lack of agreement, it is clear that successful IDEs will reduce costs due to effects such as increased process discipline.

A final point relevant to the role of IDEs in the development process was provided by our academic respondent. In response to a question regarding whether any IDE-related issues should be off-limits, he recommended that these experiments should not be used to downselect technology. This appears to be inconsistent with the notion that improved methods of evaluation enable Sponsors to make better decisions regarding the allocation of funds. However, his advice provides valuable IDE-related insight — experiments should occur earlier, rather than later, in the development process when their purpose is to evaluate technological potential and/or domain applicability. Otherwise, a downselection experiment, taking place after two or more contending approaches have already undergone lengthy development efforts, would occur too late to achieve its intended purpose of guiding a funding decision. Allowing extensive resources to be indiscriminately expended on development prior to the conduct of such an evaluation would relegate IDEs to no more than tools for deciding, in hindsight, which approaches were wasteful or had simply failed.

IDEs should act as tools to aid foresight, rather than hindsight, for it is here that they show promise in facilitating cost reduction.

This observation suggests a strategy where IDEs are performed early in the development process, in contrast to the belief expressed by many of our respondents that these experiments should be frequently conducted throughout development. Earlier, the counsel to avoid real world situations where control may be lost over events was discussed. It is interesting to note that this advice is compatible with the former viewpoint. Thus, with regard to the relationship between IDEs and the software development model being employed, the team concluded that IDEs are best carried out early in the development process; that they are best used within an iterative process; as well as that their overall effect on the model is not fully understood and will become clearer with more use.

A.5.4 Survey Conclusions

The views offered by the survey's respondents and the team's analysis of their answers permits the nature of IDEs to be clarified. This can be accomplished by identifying the necessary or desirable characteristics which these experiments must possess if they are to be successfully used to evaluate the potential use of a software technology in a specific application domain.

First, because the IDE's foremost purpose is evaluating the use of particular software technologies in an application domain, the goal of an IDE must be based on the operational needs of that domain. Other issues, such as the investigation of basic scientific principles, are secondary in this context and should not be allowed to impede the essential determination of the technology's value to the target domain's users.

Second, the team's initially proposed IDE methodology should be adapted by modifying items c. and g. of step 1, experiment specification, to incorporate the notions that:

c. The evaluation metrics to be applied should be defined by users to the greatest extent possible; and
g. The scenario(s) identified to drive the experiment must be as realistic as possible and authenticated by domain experts and/or the appropriate certifying organization(s).

Third, the underlying intent of the IDE concept, which can be summarized as experimental evaluation to guide pending and programmatic decisions, coupled with the impracticality of maintaining experimental control in real or near-real operating conditions, implies that IDEs must be performed in the earlier stages of software development. This, in turn, implies that the use of IDEs is better suited to an iterative software development model. Thus, as an example, IDEs might be incorporated into the well-known spiral model.

A.6 The IDE Definition and Its Implications

The following IDE definition is offered as a result of the questionnaire responses which were received as well as the team's experience and subsequent analysis of the relevant issues:

> *IDEs are experiments which test specific hypotheses derived from requirements reflecting operational needs and performed repeatedly throughout the development cycle.*

The definition incorporates several key ideas which were stressed by the survey respondents, including:

- Testing specific hypotheses.
- Repeated evaluation to guide development.
- Explicit requirements as the standard of evaluation.
- Operational need as the ultimate driver.

The following IDE-related conclusions augment the aforementioned definition:

- Conceptually, IDEs lie at the intersection of rapid prototyping, software experimentation, as well as verification and validation.
- IDEs are used to evaluate software intended to perform a distinct domain-related function.
- IDEs foster technology insertion, not technology advancement.
- IDEs are employed within a rapid prototyping development model to both leverage the existence of functional prototypes and avoid the additional cost of simulating functionality.
- IDEs can be conducted during numerous phases of the development cycle, including, without limitation: requirements definition and validation; design trade-off studies and design V&V; as well as integration and test.
- IDEs occur before software is exercised in the field.
- An IDE is not used as a downselection vehicle between two or more large systems or technologies primarily because it is inefficient and costly to do so. In addition, conducting IDEs earlier can avoid "sunk" costs.

- IDEs represent a special class of software experiments whose purpose is to evaluate progress toward meeting operational needs.

- IDEs yield benefits despite their costs. The benefits of IDEs include increased process discipline, clearer research and development goals, closer couplings to operational users, and improved decision making at technology and funding judgment points. IDE costs are centered around obtaining and (pre)processing authenticated data, performing repeated integrations, as well as conducting each separate IDE and analyzing the experiment's results. The disadvantages are clearly outweighed by the advantages.

A.7 Using the AAITT to Perform Instrumented Domain Experiments

Theory was placed into practice with the conduct of the Instrumented Domain Demonstration (IDD) as an example Instrumented Domain Experiment. The primary goal of the IDD was to show that the testbed could facilitate the IDE process. Therefore, as before, an existing distributed software system, in this case a Theater Missile Defense application, was embedded and integrated using the AAITT so that an IDE could be performed. This successful effort and the results obtained by the team are presented below.

The distributed application provided Battle Management/Command, Control, and Communications support for a Theater Missile Defense (TMD) Command Center. A scenario centered in the Middle East was used for the IDE. The Command Center was transitioned through three distinct modes of operation. Each mode imposed a different set of objectives on the Center. The application's decision aids were used to ultimately help the commander defend Saudi Arabia against threats launched from Iran.

The TMD application's human-computer interface was composed of five displays. The CINC Theater Command screen displayed the decision aids' assessments and was used to trigger all command functions. The Event Indicator screen showed the state of incoming threats. The Map Display screen overlaid weapon locations and missile tracks on the theater map. The Threat Assessment screen evaluated each country's threat posture. Finally, the Weapon-Target Assignment screen displayed weapon-to-target pairings.

Once embedded and integrated using the testbed, the IDD application was composed of the following six modules:

Batman..... acted as a scenario generator for the war mode of operation by propagating all weapon firings and performing weapon-to-target assignments.

Astrocalc..... predicted weapon impact points and times.

Adaptnet.....dynamically adapted various internal neural networks by reconfiguring and retraining them based on changes in weapon site locations.

Map.....provided a map of the theater. It displayed missile trajectories and allowed defensive weapon sites to be repositioned.

CINC Theater.....facilitated analysis by providing the commander with weapon and target location data. The module also allowed decisions about defensive weapon placement and civilian alerts to be introduced into the application.

Table Server.....functioned as a memory-resident data repository offering an SQL-like interface. It offered rapid query operations by eliminating the overhead associated with commercial database management systems.

The baseline IDD application architecture's topology is shown in Figure A-1. In this and subsequent figures the Batman, Astrocalc, Adaptnet, Map, CINC Theater and Table Server modules are referred to as "Batman," "Astro," "Check Nets," "CINC Theater," and "Run DB," respectively. The figures also depict the distribution of modules across host machines.

Embedding and integrating the six components using the AAITT allowed the team to gain valuable insight into the application's performance. Performance problems were detected and the database's role as a central repository suggested that it might be a source of observed bottlenecks. Initial analyses indicated that, due to the observed variability in the time spent waiting for the completion of a requested database transaction, application execution time was potentially being affected by contention among the modules for access to the database.

Using this information, the hypothesis for the example IDE was cast as:

Reducing the number of modules contending for a given copy of the database will produce a distinct effect on performance.

Two alternative architectures were subsequently designed and implemented as additional AAITT applications to test this hypothesis. These architectures duplicated the database on each host machine in the testbed, thereby reducing contention for any given copy of the repository.

The alternative architectures, shown as Figures A-2 and A-3 below, included two and three copies of "Run DB," respectively. In each case, every duplicate of the Table Server received data generated by "Batman."

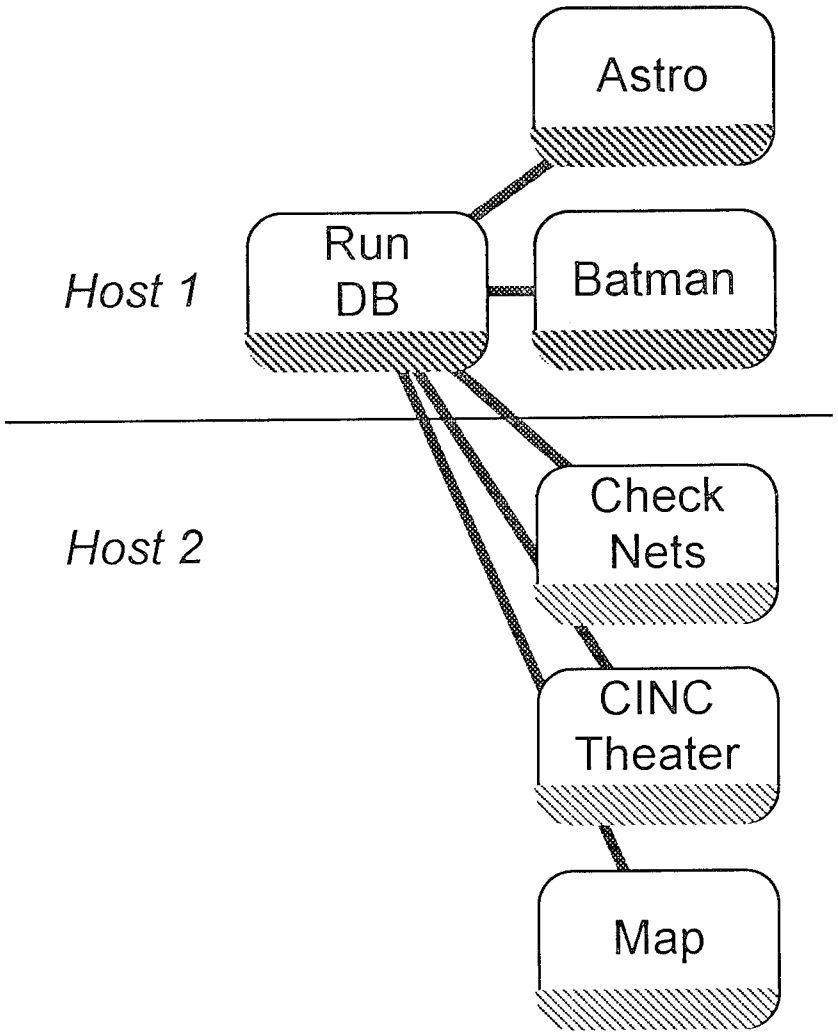


Figure A-1. TMD Application Architecture with One Database

The AAITT allowed the team to rapidly construct, execute, instrument, and monitor each of the applications. Extensive logs were captured during the execution of the baseline and alternatives. Preliminary examinations of log information related to application-level communication traffic seemed to indicate that reducing the number of modules contending for a given copy of the database did, indeed, produce a distinct effect on performance. However, quantitative results with a greater level of detail needed to be established.

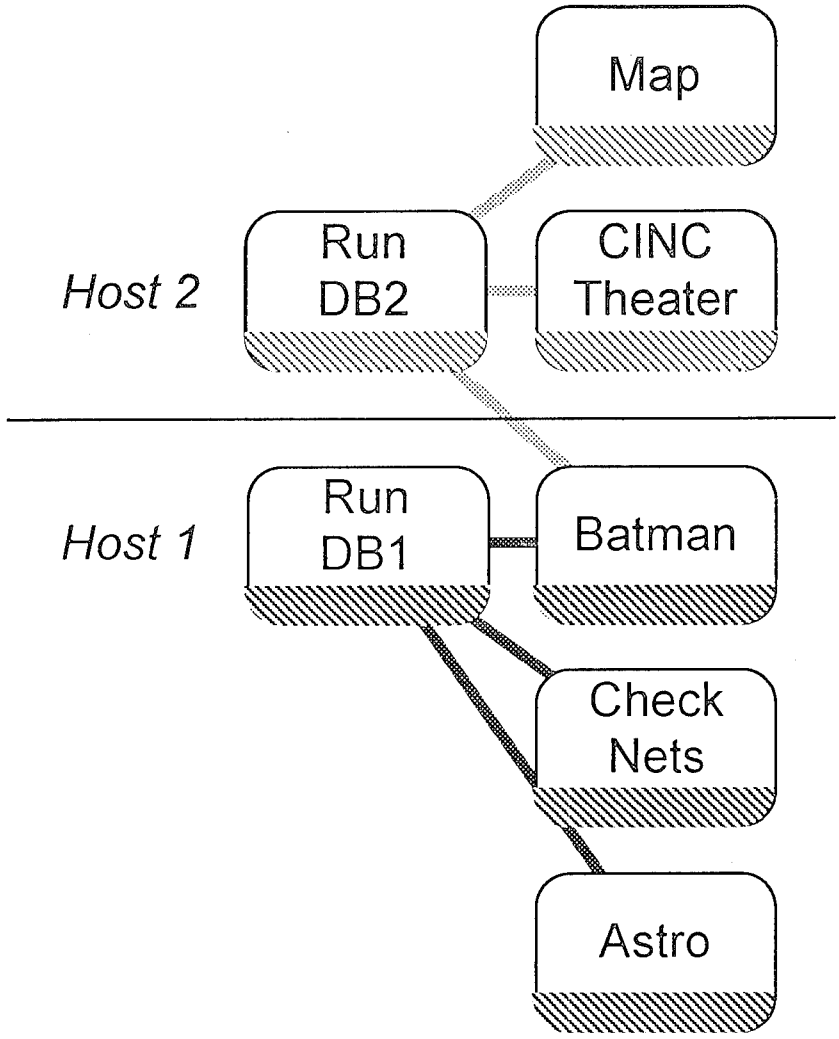


Figure A-2. TMD Application Architecture with Two Database Copies

For this reason, an analytical statistics package, the CommonLISP Analytical Statistics Package (CLASP) from the Experimental Knowledge Systems Laboratory of the University of Massachusetts at Amherst, was used to explore the data in the log files. In general, CLASP can take a dataset and calculate a large number of descriptive statistics, including the mean, median, and standard deviation. Using these statistics, tests such as the *t test* and an Analysis of Variance can be computed.

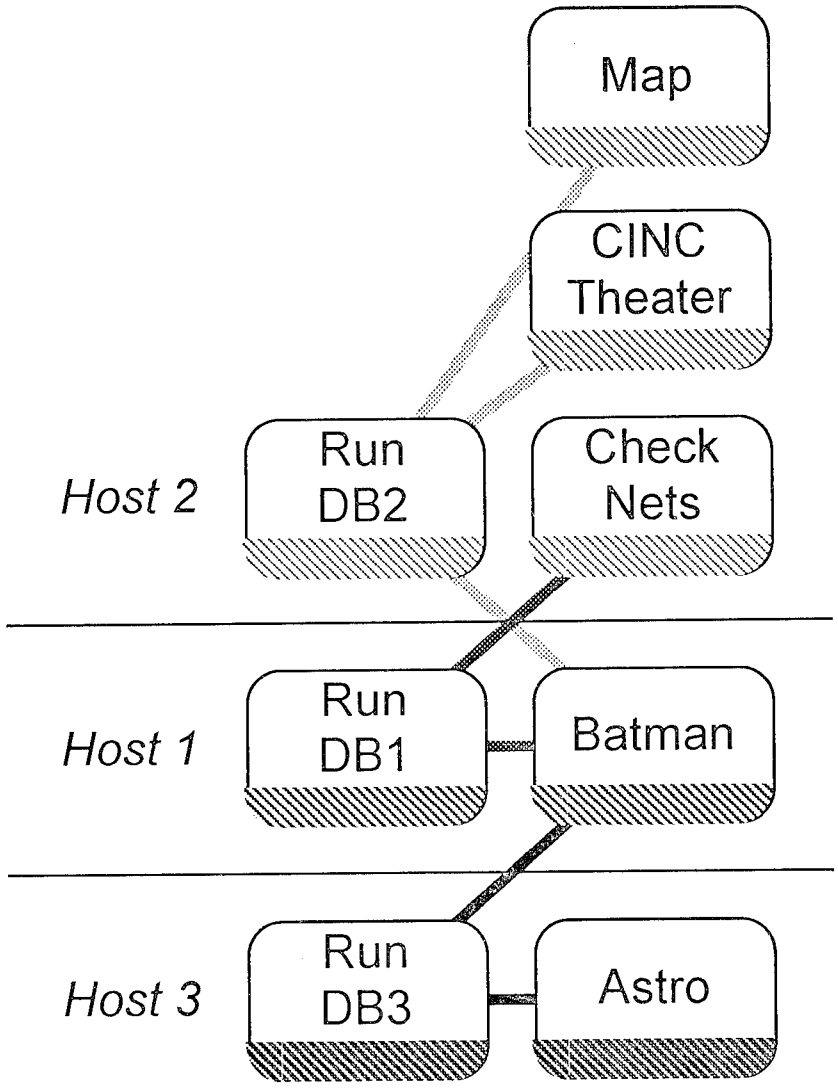


Figure A-3. TMD Application Architecture with Three Database Copies

The format of the AAITT's log files is very similar to the input format required by CLASP. Several *emacs* macros were generated to effect the transformation, namely, by eliminating unreadable LISP characters; inserting a dataset name into the log file; and introducing column, or variable, names for the data. The transformed log files were then loaded into CLASP and statistical summaries were prepared using the package. From these summaries, statistics describing

database queries were used as the foundation of the two graphs shown in Figures A-4 and A-5. Note that in both figures, the terms "tmd_good," "tmd2_good," and "tmd3_good" along the x-axis representing "Architecture" refer to the topologies presented in Figures A-1, A-2, and A-3, respectively.

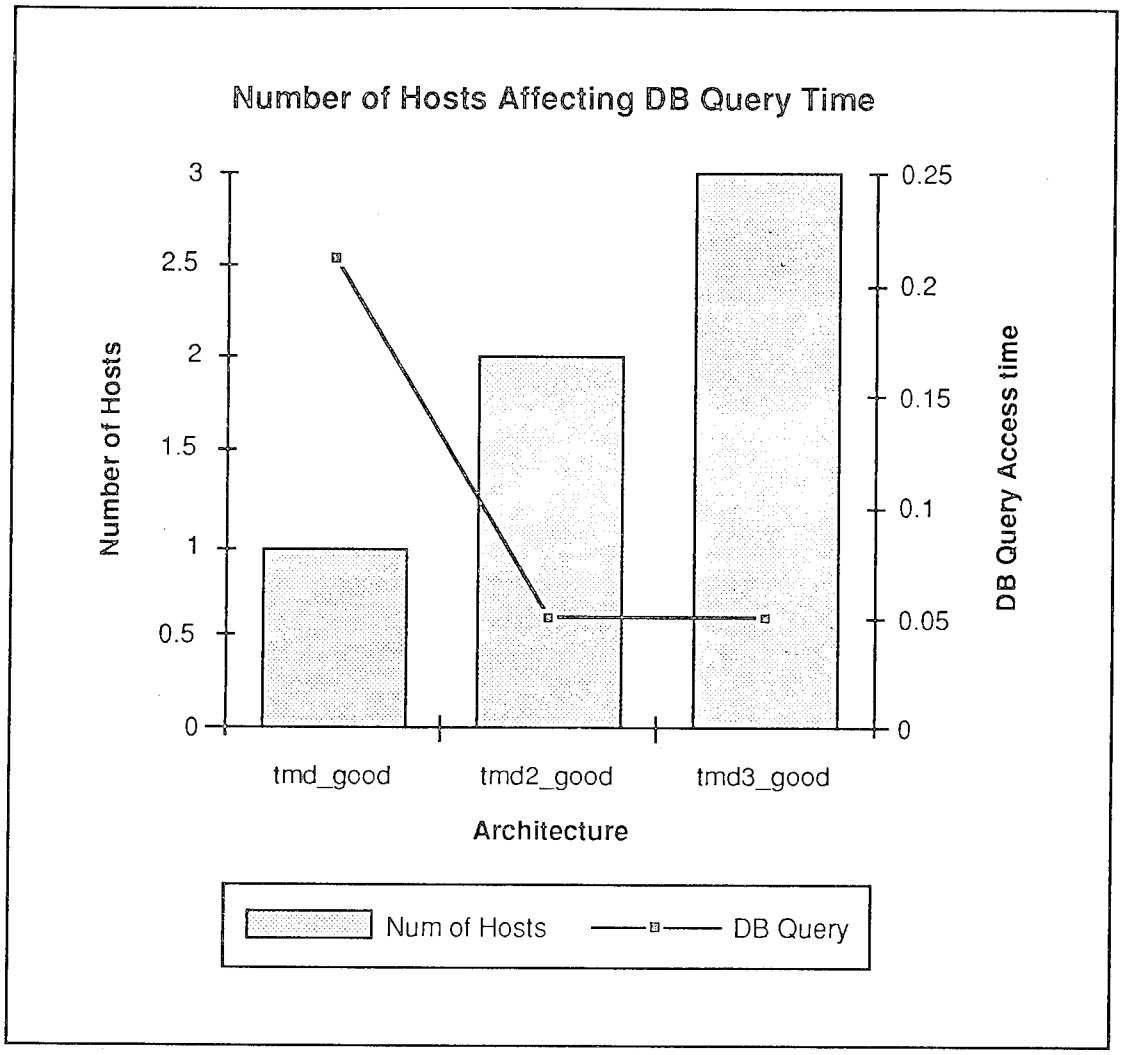


Figure A-4. Number of Hosts Affecting DB Query Time

The graph of Figure A-4 shows that, as the database is replicated (number of hosts greater than 1), the database query time drops from 0.21 to 0.05 seconds. The observed fourfold speedup confirmed the team's hypothesis. Figure A-5 shows a latent effect of reduced query time as the time between queries grows.

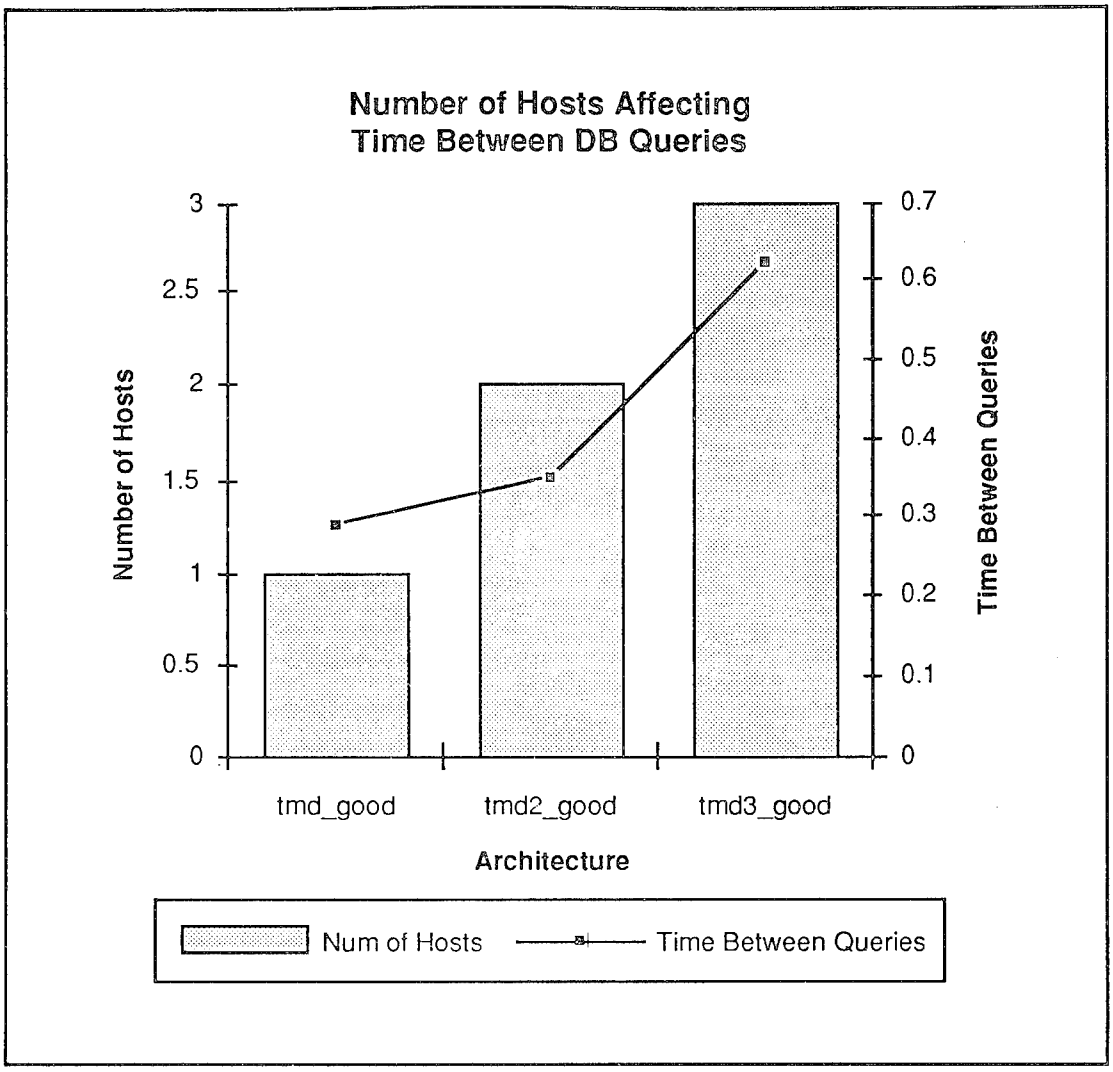


Figure A-5. Number of Hosts Affecting Time Between DB Queries

Table A-2, below, summarizes the application of the team's Instrumented Domain Experiment methodology to realize the Instrumented Domain Demonstration, a specific instance of an IDE. The table's left column presents each of the methodology steps introduced earlier in section A-3 and the entries found in the right column list the corresponding task(s) completed by the team to prepare the IDD.

IDE Methodology Step	IDD Development Task(s)
1. Produce the specification of a particular experiment.	◇ The TMD application was identified for the IDE. ◇ Investigating the effect of database contention on performance was established as the goal. ◇ A scenario centered in the Middle East was selected. ◇ The AAITT's instrumentation and monitoring capabilities were used as the measurement and data-capture strategy. ◇ "DB Query Time" was the experiment's evaluation metric.
2. Introduce the application suite into the instrumentation facility by embedding and integrating the application's constituent components.	◇ The TMD application's original, custom communication layer was removed. ◇ The AAITT's Frameworks were used to successfully embed and integrate the problem-solving suite.
3. Implement or integrate the experiment's required data-capture mechanisms.	◇ The testbed's built-in measurements were sufficient to capture the required data.
4. Conduct the experiment under controlled conditions.	◇ The MCM Workstation's features permitted experiments to be conducted in a controlled and repeatable fashion.
5. Analyze the data captured during the conduct of the experiment.	◇ Logged data was initially analyzed using the AAITT's Metrics Analyzer, and later examined within CLASP.
6. Present analytical results to the targeted audience for review.	◇ Results were presented to RL/C3CA during the IDD.

Table A-2. Realizing the IDD by Applying the IDE Methodology

A.8 IDE Support Provided by the AAITT

The Instrumented Domain Demonstration was auspiciously completed. The feasibility of using the AAITT as a foundation for conducting Instrumented Domain Experiments was convincingly confirmed. In addition, the value of an environment for the design, analysis, integration, evaluation, and execution of large-scale, complex, distributed software systems was strongly revalidated. The entire process of embedding and integrating the Theater Missile Defense decision support suite, including application familiarization and replacement of the communication layer, required only approximately two staff-weeks of effort. Constructing alternative architectures in support of the experiment was effortlessly accomplished once the embedded components were catalogued as AAITT modules and available for adaptation and reuse. Realizing the IDD application and conducting the Experiment with the AAITT led the team to conclude that the testbed strongly supports IDEs in a variety of ways. Table A-3, below, highlights these facts. The table's left column reiterates the IDE methodology. Summaries of the relevant testbed capabilities supporting each step appear in the right column.

The AAITT's capabilities were also extended with the development of an integrated approach to effecting detailed data exploration using CLASP. A graphical overview of this process is presented in Figure A-6.

IDE Methodology Step	AAITT Support Feature(s)
1. Produce the specification of a particular experiment.	◆ This portion of the IDE methodology is primarily a planning step. However, the testbed's cataloguing capabilities promotes reuse by allowing users to easily compare what they already have with what they need.
2. Introduce the application suite into the instrumentation facility by embedding and integrating the application's constituent components.	◆ The MCM Workstation's modeling, code generation, and compilation capabilities allow components to be rapidly and cost-effectively embedded, integrated, and (re)configured. ◆ Extensive distributed system debugging tools facilitate the realization of properly-functioning solutions.

3. Implement or integrate the experiment's required data-capture mechanisms.	◈ The AAITT provides a wide assortment of built-in measurements for any application. ◈ The development and insertion of custom, user-defined measurements is also supported to meet situation-specific needs.
4. Conduct the experiment under controlled conditions.	◈ Application execution follows a strictly-enforced state transition strategy providing control and repeatability. ◈ Monitoring, or execution-time data-capture, is a non-intrusive facility that minimizes the introduction of false artifacts. ◈ Flexible, menu-driven host (re)assignment facilitates the investigation of resource usage and communication issues.
5. Analyze the data captured during the conduct of the experiment.	◈ The AAITT's Metrics Analyzer and Log Viewers both provide facilities for examining log data in both graphical and tabular forms. ◈ The log data is well-formed and maintained in the ASCII format. Thus, the data can be easily translated into other formats required for incorporating it into analysis tools such as CLASP or spreadsheets.
6. Present analytical results to the targeted audience for review.	◈ The graphs created by the Metrics Analyzer are easily understood and suitable for incorporation into technical presentations.

Table A-3. AAITT Support for Instrumented Domain Experiments

In this figure, a testbed-resident application is constructed, instrumented, executed, and monitored to collect data logs. These data logs may first be viewed with the AAITT's Metrics Analyzer. The logs can then be transformed, introduced into CLASP, and analyzed to uncover latent information. The user, an Application Architect, then uses the information from the analysis to further refine the application. This process parallels the iterative nature of IDEs.

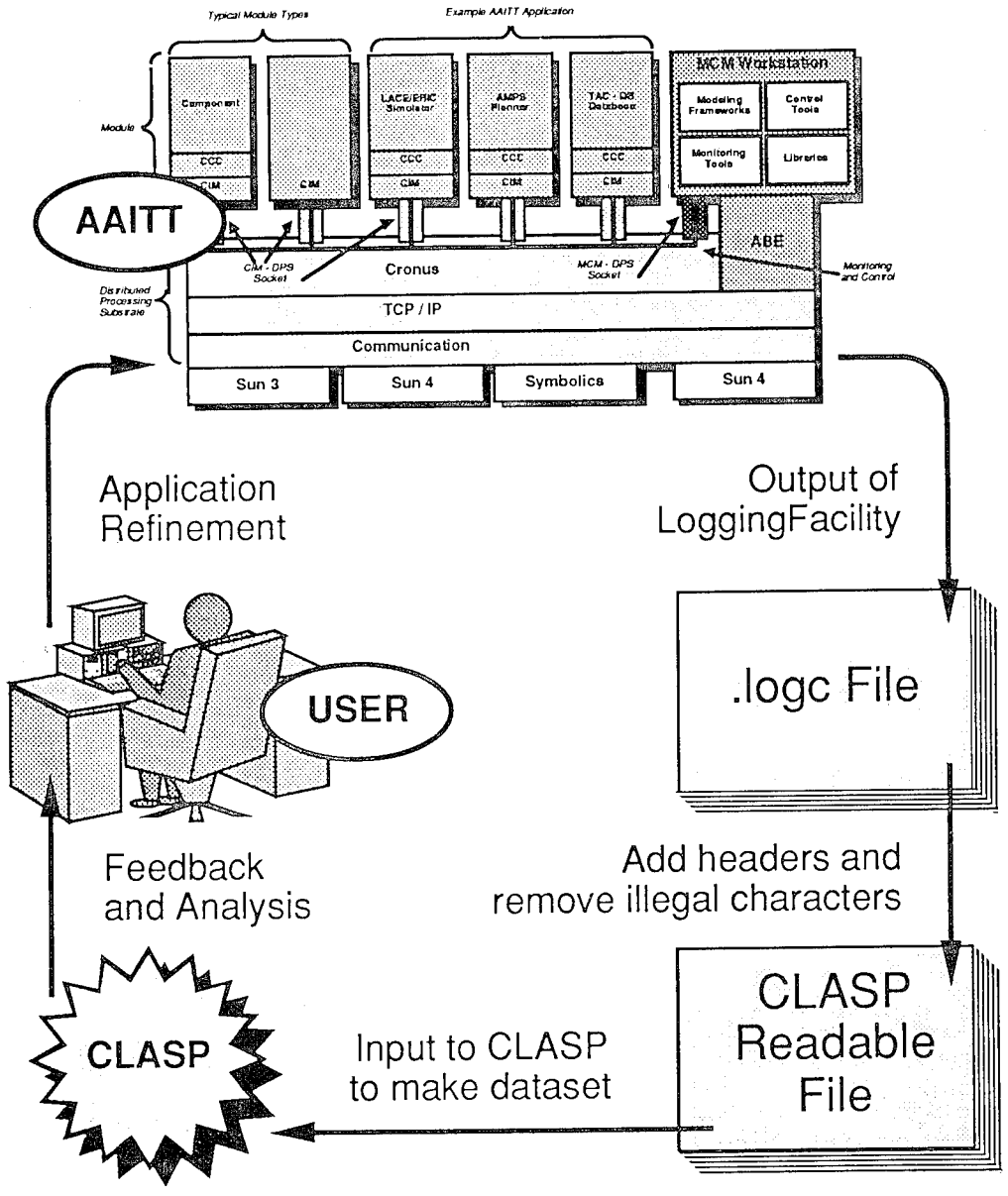


Figure A-6. Using CLASP for Data Exploration to Complement AAITT

Glossary

Allowable AAITT States	The permissible modes in which an AAITT application or module can be found. The allowable states are: Unloaded, CIM Loaded, CIM Connected, Loaded, Initialized, Running, and Paused.
Application	An assemblage of modules and/or subapplications.
Application Architect	The user role responsible for defining the multi-module application and determining how modules interact.
Asynchronous	In the context of AAITT, the type of concurrent processing where process completion is not predetermined.
Breakpoint	Processing instruction that is triggered by the occurrence of a specific predetermined condition. This processing instruction suspends program execution.
Channel	A single row of data within a time-dependent measurement display.
CIM Connected	An allowable AAITT module state defined as one where communications have been established by the module with the DPS. In this state the component of the module is not in main memory.
CIM Loaded	An allowable AAITT module state defined as one where a module's CIM is in main memory but the component part of the module is not.
CIM Reset	An allowable AAITT module state transition defined as placing a module from the CIM Connected, Loaded, Running, or Paused state to the CIM Loaded state.
Code Frame	Skeletal Component Interface Manager code requiring subsequent customization.
Code Frame Generation	The automated production of a code frame based on a user-specified graphical model.

Component	A stand-alone conventional or knowledge-based program.
Component Embedder	The user role responsible for transforming an independent component into an AAITT-compatible module.
Component Interface Manager	A software element responsible for managing the interface between a component and the rest of the testbed.
Connect	An allowable AAITT module state transition defined as placing a module from the CIM Loaded state to the CIM Connected state.
Conventional Software	Software that does not utilize any special Artificial Intelligence techniques or use any special knowledge representations.
Database Management System	A software system that makes uses of a specially designed language and logical structure for data that optimizes data organization and access.
Distribute	An allowable AAITT module state transition defined as placing a module from the Unloaded state to the CIM Loaded state.
Dynamic Message	A data structure which is passed between the message template generating facility and an AAITT entity to test communication interfaces.
Execute	An allowable AAITT module state transition defined as placing a module from the Initialized state to the Running state.
Filter	Program that takes, as input, a measurement log and produces, as output, a subset of the log expressed as a data structure containing only the desired features of the original log, as specified by the user.
Filtering	The act of applying a filter to a measurement log.

Graphical Application Model	The representation of an AAITT application in the form of a schematic diagram showing the application's constituent modules and their respective port connections.
Initialize	An allowable AAITT module state transition defined as placing a module from the Loaded state to the Initialized state.
Initialized	An allowable AAITT module state defined as one where the module is in main memory, the CIM is connected to the DPS, and pre-run inter-module messages have been sent (e.g. mission-context has been established).
Instrumentation	Used to display measurements.
Knowledge-Based Software	Software that makes use of inductive or deductive reasoning about the data it has access to in order to form hypothesis or reach conclusions.
Load	An allowable AAITT module state transition defined as placing a module from the CIM Connected state to the Loaded state.
Loaded	An allowable AAITT module state defined as one where the module is in the memory of an AAITT machine.
Log	Data structure containing time-stamped records of user-designated measurements that is collected during application execution.
Logging	The act of accumulating a log. The AAITT provides a facility for the creation of logs.
Long-Form Logs	Time-stamped records of user-designated message flows that are collected during application execution.
Measurement	Quantifies features that aid in the understanding of system performance.
Measurement Logs	Time-stamped records of user-designated measurements that are collected during

	application execution. A Measurement Log is also referred to as a Short-Form Log.
Message	A data structure passed between any two AAITT entities to effect communication.
Mixed State	An AAITT application mode in which all constituent modules are not in the same allowable AAITT state.
Modeling	The act of specifying an AAITT graphical representation or structure.
Module	A component (possibly modified) plus its associated component interface manager.
Monitoring	The set of procedures through which measurements are captured so that instrumentation can be applied to them.
Operating System	The layer of software that controls resources and hardware. This layer may also direct firmware. The operating system supports and is utilized by both application software and users.
Paused	An allowable AAITT module state defined as one where the module has been taken out of the running state (whether by self-breakpointing, external suspension, or due to an error) and can be returned to the running state directly, without having to enter any other state first.
Port	A communications interface point for an AAITT module.
Relational Database	A data organization where the data is made up tables. Each table is comprised of records. Each record in the same table has the same logical fields. Data in two or more tables can be joined if they have one or more related fields.
Remote Procedure Call	One of the AAITT-supported methods of invoking an operation or process. It is

	essentially a "call and block until complete" style of invocation.
Rendezvous	One of the AAITT-supported methods of invoking an operation or process. It uses a "futures" paradigm of calling that provides a "call and don't block" style of invocation, where a "handle" is provided for each invocation. The "handle" can be claimed at some future time to verify that the invoked operation has completed.
Reset	An allowable AAITT module state transition defined as placing a module from the Initialized, Running, or Paused state to the Loaded state.
Resume	An allowable AAITT module state transition defined as placing a module from the Paused state to the Running state.
Running	An allowable AAITT state that signifies that either the application or a specific module is executing program instructions.
Self-Test	The ability of an entity to determine its own status. For AAITT, this term refers to the testbed software's ability to determine if it is installed correctly.
Short-Form Logs	See Measurement Logs.
State Restore	The act of placing an AAITT application or module in a state that had been previously saved.
States	The conditions or modes in which AAITT applications or modules can exist..
State Save	Saving the current state values of a CIM to a file for later CIM state restoration.
State Transitions	The passage from one allowable AAITT state to another.
Step	An allowable AAITT module state transition defined as placing a module from the Paused

	state to the Running state, allowing the module to either receive one application message or send one out, and then place the module back into the Paused state.
Subapplication	An assemblage of modules and/or subapplications.
Suspend	An allowable AAITT module state transition defined as placing a module from the Running state to the Paused state.
Suspending	Placing the application into a paused state from the running state.
Synchronous	In the context of AAITT, the type of concurrent processing in which the order of process completion is predetermined.
Terminate	An allowable AAITT module state transition defined as placing a module from any allowable AAITT state, except Unloaded, to the Unloaded state.
Testbed	Facility that provides tools for experimenting with software system configurations in order optimize performance and solutions.
Unload	An allowable AAITT module state transition defined as placing a module from the Loaded, Initialized, Running, or Paused state to the CIM Connected state.
Unloaded	An allowable AAITT module state defined as one where the module is not in the main memory of any AAITT machine.
Unloading	Removing an entire application (all modules and associated component interface managers) out of the main memories of all AAITT computers.
Versioning System	Facility provided by an operating system that enables the tracking of file creation and modification.

Abbreviations and Acronyms

AAITT	Advanced Artificial Intelligence Technology Testbed
ABE™	A Better Environment
AFWL	Air Force Wright Laboratory
AI	Artificial Intelligence
AMPS	A Mission Planning System
ARPA	Advanced Research Projects Agency
ARPI	ARPA/Rome Laboratory Planning Initiative
ATO	Air Tasking Order
BBN	Bolt, Beranek, and Newman
CIM	Component Interface Manager
CLASP	CommonLISP Analytical Statistics Package
CORBA	Common Object Request Broker Architecture
COTS	Commercial-off-the-Shelf
CPU	Central Processing Unit
CSCI	Computer Software Configuration Item
C3I	Command, Control, Communications and Intelligence
DART	Dynamic Analysis and Replanning Tool
DBMS	Database Management System
DoD	Department of Defense
DPS	Distributed Processing Substrate
ERIC	Extensions to Ross In Common LISP
FIFO	First In, First Out
FMERG	Force Module Enhancer and Requirements Generator
GFS	Government Furnished Software
GOTS	Government-off-the-Shelf
GUI	Graphical User Interface
IEEE	The Institute of Electrical and Electronics Engineers
LIFO	Last In, First Out
LISP	List Processing
MCM	Modeling, Control and Monitoring Workstation
N/A	Not Applicable
OSD	Office of the Secretary of Defense

R&D	Research and Development
RFP	Request for Proposal
RL	Rome Laboratory
SAM	Surface to Air Missile
SOCAP	SIPE for Operations Crisis Action Planning
SQL	Structured Query Language
SSS	System / Segment Specification
SRS	Software Requirements Specification
TAC-DB	Tactical Database
TCP / IP	Transmission Control Protocol / Internet Protocol
TMD	Theater Missile Defense
USAF	United States Air Force
WAN	Wide-Area Network

*U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20105

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.